# A formal model for strategic planning in cooperative and competitive environments Case study: design and implementation of a basketball simulator

Guilherme Fernandes Otranto

A Thesis submitted
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
for the Degree of
Doctor in Computer Science

Doctoral Program: Computer Science

Advisor: Prof. Dr. Junior Barrera

São Paulo, July 2017

**A formal model for strategic planning in cooperative and competitive environments Case study: design and implementation of a basketball simulator**

Esta versão da tese contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 13/09/2017. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Junior Barrera (orientador) - IME-USP
- Prof. Dr. Luiz Carlos Pacheco Rodrigues Velho - IMPA
- Prof. Dr. Eduardo Aoun Tannuri - EPUSP
- Prof. Dr. Leonardo Lamas Leandro Ribeiro - UnB
- Prof. Dr. Claudio Santos Pinhanez - IBM - Reasearch-SP

# Acknowledgments

I would like to begin by thanking all my professors at IME for the amazing work they did and the support I received during my time there. From the first time I entered the building in 2004 up to now I could always count on the deep knowledge and willingness to help from the faculty members.

A special thanks to professor Junior Barrera whose guidance made this work possible. We started working together in 2006 and through the years we have been able to accomplish much. I value immensely all that I learned in our time working together. His targeted and insightful advice helped shape this document and our work into what I believe to be a truly remarkable accomplishment.

I would also like to thank Professor Leonardo Lamas, who was an essential member of our team and helped us shape the basis of our work into a solid and robust foundation. His insights into the nature of sports and human players gave our work a much needed real world knowledge base.

I am also immensely grateful to the members of the judging commission and Prof[a]. Dr[a]. Leliane Nunes de Barros for their insightful feedback during the presentation of this thesis.

Finally, I would like to thank my family, friends and business partners for their support and understanding as I tried to balance my social life, work and study. Founding and working on a company while actively pursuing the completion of this thesis was a grueling task that would have certainly been impossible to achieve without their support.

# Abstract

OTRANTO, G. F. **A formal model for strategic planning in cooperative and competitive environments. Case study: design and implementation of a basketball simulator**. 2017. Phd thesis - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2017.

The motivation that originated this work was the desire to create an invasion team sports simulator capable of applying user defined strategies to guide the behavior of the agents in the simulation. With this objective in mind we created a formal strategy model to describe complex team behavior and developed methods of using that model to calculate collective plans. We defined both the strategy model and the planning methods in a broad manner that can be applied in many different domains. Then we defined a basketball simulation domain and implemented our methodology to develop a simulator. We also present a control system architecture that is compatible with our proposed planner and show how we implemented it to create the basketball simulator.

The formal strategy model we developed can be used to represent team behavior, analyze real world events and create simulations. We developed a strategy design tool that allows the end user to create and visualize team strategies for basketball. Finally, we developed a system that interprets the user generated strategies and creates a basketball match simulation of the described behavior.

We also proposed a methodology for the development of simulation systems involving multiple intelligent agents. Our recommended control system architecture separates the many layers of control, which simplifies the development process and results in a naturally expansible system.

In this thesis we have provided a novel approach to collective behavior simulation utilizing user input as a guide to the strategy planning. Both the theory and methods developed have been tested through the implementation of a basketball simulator and the results were satisfactory. We believe this is a seminal work that will lead to many interesting developments, both in the realm of sports and in broader domains.

**Keywords:** planning, behavior simulation, artificial intelligence.

# Resumo

OTRANTO, G. F. **Um modelo formal para planejamento estratégico em ambientes cooperativos e competitivos. Estudo de caso: desenho e implementação de um simulador de basquete**. 2017. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2017.

A motivação que deu origem a esse trabalho foi um desejo de criarmos um simulador de esportes de invasão coletivos capaz de aplicar estratégias definidas pelo usuário para guiar o comportamento de agentes na simulação. Com esse objetivo em mente nós criamos um modelo formal de estratégia para descrever comportamentos complexos em equipe e desenvolvemos métodos para usar esse modelo no cálculo de planos coletivos. Definimos o modelo e os métodos de planejamento de uma forma abrangente que pode ser aplicada em muitos domínios diferentes. Definimos um domínio para a simulação de partidas de basquete e implementamos nossa metodologia para desenvolver um simulador. Também apresentamos uma arquitetura de controle que é compatível com o planejador proposto e mostramos como implementá-la na criação de um simulador de basquete.

O modelo formal que desenvolvemos pode ser usado para representar comportamento coletivo, analisar eventos reais e criar simulações. Desenvolvemos um desenhador de estratégia que permite que o usuário final desenhe e visualize estratégias de equipes de basquete. Finalmente, desenvolvemos um sistema que interpreta o conteúdo gerado pelo usuário e cria uma simulação de basquete usando o comportamento descrito.

Propusemos também uma metodologia para o desenvolvimento de sistemas de simulação envolvendo múltiplos agentes inteligentes. Nossa arquitetura de controle separa as várias camadas de controle, simplificando o processo de desenvolvimento e resultando em um sistema naturalmente expansível.

**Palavras-chave:** planejamento, simulação de comportamento, inteligência artificial.

# Contents

# List of abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| MDP | Markov Decision Process |
| FSM | Finite State Machine |
| BT | Behavior Tree |
| IK | Inverse Kinematics |
| TSM | Team Strategy Model |
| SM | Strategy Map |
| ST | Strategy Transition |
| GUI | Graphical User Interface |
| MCTS | Monte Carlo Tree Search |

# List of Symbols

| | |
|---|---|
| $\mathbb{N}$ | Set of all natural numbers |
| $\mathbb{I}$ | Set of all integer numbers |
| $\mathbb{R}$ | Set of all real numbers |
| $\mathbb{R}^*$ | Set of all non-negative real numbers |
| $\mathbb{R}^+$ | Set of all positive real numbers |
| $\{e_1, e_2, ..., e_n\}$ | A list of $n$ elements, named $e_1$ through $e_n$ |
| $\in$ | Is an element of |
| $\notin$ | Is not an element of |
| $\subseteq$ | Is a subset of |
| $\forall$ | For all |
| $\neg$ | Negation of a logical proposition |
| $[x, y]$ | The interval between $x$ and $y$, including both |
| $[x, y)$ | The interval between $x$ and $y$, including $x$ and excluding $y$ |
| $(x, y]$ | The interval between $x$ and $y$, including $y$ and excluding $x$ |
| $(x, y)$ | The interval between $x$ and $y$, excluding both |
| $<_G$ | Smaller than according to the order of ordered set G |
| $min(x, y)$ | The smallest value between $x$ and $y$ |
| $max(x, y)$ | The greatest value between $x$ and $y$ |
| $\pi$ | A policy in a probabilistic planning problem |
| $\gamma$ | The reward value of a strategy map |
| $\omega$ | The risk value of a strategy transition |
| $\mu$ | Metric to compare distance between a SM and a domain state |
| $\psi$ | Scoring function to rank plays |

# Chapter 1

# Introduction

When we first considered tackling the challenge of creating a sport simulator that obeyed the user defined strategy, we identified two main hurdles: how could we represent this complex knowledge and how can we use this representation to guide a simulation. In 2006 we started to investigate exactly how big of a challenge that would be. We decided to start by creating a very simple domain as a stepping stone towards our main goal of simulating a sportive match. We defined a turn-based soccer game inspired by the old button soccer game and called it Strategy Soccer. The turn-based nature of the game allowed us to pause the simulation to request user input, which removed some of the biggest control challenges. By 2008 we had achieved a promising result: we had a simple soccer simulation where the user could provide some strategy tips that would automatically control some of the players. We still depended heavily on user input, but the essence of an automatic control system seemed to be there. Around that time Leonardo Lamas joined our team and we changed our focus to a more generic approach that could be used in any invasion team sports. Due to his specialty with basketball we ended up choosing that as our case study. By the time I started this thesis we already had a strong conviction that a strategy model could be formalized and that we could use it to develop a novel kind of sport simulator.

Our intention is to simulate user defined behavior, not to find the best possible action to suit some situation. This is the main difference between our work and other AI frameworks that aim to calculate the best actions to reach the best outcome (e.g., satellite automatic control). How the behavior of an AI agent is perceived by humans can be a contributing factor in some AI uses, as [WSN+17] quite recently pointed out. Systems to control satellites or factory robots might not require the behaviors generated to be consistent to that of a human, but many uses of AI actually benefit from a more human behavior, such as enemy AI in video games or real world simulations. We aim to simulate the strategy defined by the user, including any potential flaws in his reasoning.

The domain of invasion team sports (e.g., basketball, soccer) is too complex to simulate using traditional AI planning techniques. This is in great part due to the fact that an infinite number of possible situations can arise from any match between two teams, meaning that the state space for the planning is infinite. Furthermore, the actions in this domain are not easily describable and the outcome is never certain. Our model accounts for both those problems, allowing the planner to work with a finite (and usually small) state space and actions that represent complex behaviors and deal with uncertainty. With this model we managed to apply existing AI techniques to calculate and implement plans to any situation on complex and previously unexplored domains. The idea of transforming a planning problem into a simpler one has been explored before ([BF97]), but our approach does so in order to provide the user with the capacity to specify the desired behavior of the agents, not simply to achieve performance improvements (though that is a fortunate collateral effect).

Our first step was the specification of a formal model to describe team strategy. This model is described in [LBOU14] and it serves as a bridge between the sport specialists (e.g., managers, analysts) and the simulation software. The model actually defines a grammar that can be used to specify complex strategy which can be interpreted by the simulator. This thesis presents an

evolution of our published model ([LBOU14], [LSOB14] and [LOB16]) that generalizes both the strategy representation and the compatible planning methods.

While the concepts presented here are largely generic, we choose to focus our examples and our own implementation of the software on basketball. This choice is due to our own capabilities and to the fact that basketball has complex and well-know strategy models which we can use to ensure that our model has sufficient representative power.

We reduce the original problem of planning in a continuous domain to a simpler one by using equivalence classes. We manage to represent a continuous space using just a few representative classes to group relevant situations. We allow that an arbitrary but limited number of classes be defined when describing some desired behavior. We then use our proposed methods to transition between the original problem and this equivalence class representation. This method allows us to plan for any situation in the original domain by planning on our smaller and more manageable representation.

We also created a control system architecture that is divided in several layers. A high level control is achieved by the first layer and each layer after that will improve on the result of the previous layer by adding detail and sophistication to the behavior. The final layer actually deals with the mechanical control of each agent, such as limb placement, to provide a detailed rendering of the simulation to the user. This architecture fits every level of control in a coherent structure, from collective strategic actions all the way down to hand and foot placements. Each layer in the system can work independently and this allows for an incremental sophistication of the whole by improving on any layer individually.

## Objectives

Our goal was to create a realistic sport simulator that accepted the user's input as a guide to the collective behavior. We wanted the simulation to be as realistic as possible both in the mental behavior of the players and the physical realization of that behavior. We also wanted to provide an architecture that would allow such a system to be created in a modular manner that is compatible with future increments and sophistication.

We wanted to provide a complete solution to the problem we proposed to tackle. Since user input played a big part in any solution, we needed to quickly choose a concrete domain so that we could gather user input to validate and improve our ideas. We defined basketball as our domain sport early on and we have worked closely with basketball specialists to determine what were the essential parts of the model and what priorities should be followed. It is worth noting that a principal concern during the development was ensuring that the methods and techniques created were generic enough to be applicable outside of basketball or even sport simulation.

The reliance on user input and validation created a need for users, so both the formal model and the simulator needed to be comprehensible and usable by our target users. Therefore, we could not require deep mathematical or computational knowledge from our users for our system to work properly. To ensure maximum adoption we needed to implement user facing tools that were easy to use and we needed to be able to deploy to all common user devices and platforms.

## Document Outline

The next chapter of this document will discuss the state of the art of the areas of computer science that were used during the development of our work. Then we will dedicate the following chapters to explain our formal model, the planning techniques we created and the system architecture we propose to implement it all. Finally we will use the last chapters to show the result of our implementations and discuss the work presented.

Chapter 2 will discuss the fundamentals and the state of the art of the areas we relied on during the development of this thesis. Section 2.1 will discuss the planning framework which we used to implement the top layer of our control system: team planning. This section will cover the classical

planning problem (2.1.2), a more representative manner of describing the domains with factored representation (2.1.3) and end with probabilistic planning (2.1.4). Next, in section 2.2, we will discuss the main techniques available for individual behavior simulation. Namely we will discuss finite state machines (2.2.1) and behavior trees (2.2.2). Finally, in section 2.3 we discuss the main techniques available for animation control and mechanical simulation.

We will start to describe our contributions in chapter 3 with the formal strategy model. We divided this chapter in three sections: the formal definition (3.1), the basketball's offensive strategy model (3.2) and the basketball's defensive strategy model (3.3). The main components of the strategy models are covered in 3.1, specifically, the strategy map (3.1.1), the strategy transition (3.1.2) and the strategy graph (3.1.3). We also present an equivalent representation that can enhance individuality in 3.1.4 and a method for real world analysis using our model in 3.1.5.

In chapter 4 we show how we used the model to implement our top level planning. This chapter is divided in two sections: the formal definitions (4.1) and our implementation in the basketball domain (4.2). The formal definitions section describes the planning process in three steps: the matching function (4.1.1), the planning (4.1.2) and the realization function (4.1.3).

Chapter 5 describes our proposed architecture in four sections, one for each layer. The collective planner layer is described in section 5.1, the individual planner in section 5.2, the physics simulator in section 5.3 and the mechanical controller in section 5.4. We discuss the our own implementation of this architecture to create the basketball simulator in chapter 6.

Finally, we will discuss our results, contributions and planned future work in chapter 7. And, to reiterate, our contributions are contained in chapters 3 through 6.

# Chapter 2

# State of the Art

This chapter will discuss the methods and techniques available in literature that we used to develop our work. In 2.1 we will cover planning, which provided the basis for some of our formal model representation and how we use it in the collective planner. In 2.2 we cover the behavior implementation techniques we relied on when designing our individual planner. Section 2.3 will cover the main techniques available in literature that allowed us to acquire, synthesize and adapt the necessary animations to properly control the agents' bodies during simulation.

## 2.1 Planning

We use planning to refer to the AI area that deals with the calculation of plans (sequences of actions) that can be executed on some domain by an intelligent agent. Planning is defined in this very generic manner because it was created to handle problems in a generic way. The typical planning solver receives the description of the world (i.e., actions and possible states) as an input, as well as the specification of the problem it should solve (i.e., initial and goal states). Classical planning and its many variations has been widely studied in AI literature ([Cas98], [RN09]). On of the main advantages of planning is that a solver can work on many domains, some not even considered by the original programmer ([HTD90]). However, this capacity to work on many domains comes at a substantial cost to performance, which can sometimes be mitigated by training the solver with domain specific examples ([CCO+12]). A contributing factor to this cost in performance is the fact that one cannot use specific heuristics due to lack of domain knowledge. There are two areas in which efforts are being dispersed in planning: the first is to broaden the classes of problems that can be solved by the planner and the second is to improve planner performance.

In its most essential form, a planning problem can be described as an agent who can manipulate his environment though actions. The environment can have many different configurations (hence referred to as *states*) and actions performed by the agent change the state of the environment. There is an initial state and a group of desired states, so the planner's job is to encounter the sequence of actions that can take the agent from that initial state to a desired one. Finally, we also consider that each action has a cost and add to the planner's job that the sequence he finds should also minimize cost.

In the original approach to planning, described in subsection 2.1.2, there are two important assumptions over the domain:

- discrete: there is a finite number of possible states in the environment, and these are seen as black boxes, the user know nothing about their nature
- deterministic: every action has a predictable, deterministic outcome

It will be useful to our work to relax both these assumptions. In subsection 2.1.3 we will see how a more representative description of the states can improve planner performance. While not quite eliminating the discrete assumption, it is an important stepping stone towards that goal. In

subsection 2.1.4 we will see how to model probabilistic actions, which will be useful when we begin modeling actions with some unknown outcomes.

The amount of work being done in planning, both to expand its possible uses and to improve the performance, is an indicator of how powerful and useful planning can be. The fact that a domain can be described by the end user makes planning very attractive when one considers using specialist knowledge to guide the AI. In this thesis, for instance, we will define a planning problem where the domain is some user defined strategy we wish to simulate.

### 2.1.1 Problem Definition - STRIPS and other languages

A domain and problem specification language is necessary for planners to be truly generic in nature, receiving both environment description and the agent's task though input. The first of such language to be developed was called STRIPS (STanford Research Institute Problem Solver) and it allowed users to create and solve new problems in existing planners ([FN71]). While not exactly user friendly in today's standards, the STRIPS language is much simpler than actually programming a specialized solution. This language creation approach ended up shaping much of what is done today in planning ([FN93]). Each domain expansion and new feature in planners came coupled with a new and more representative language ([MGH+98], [FL03], [GL05]).

### 2.1.2 Discrete and Deterministic Planning

The original planning formulation is a very useful basis to understand the current state of the art planners. This formulation is sometimes called classical planning ([RN09]). While the formulation is quite simple, many useful and insightful ideas can be traced back to this simple origin. It is also worth noting that we lifted the graphical representation of the problem directly from this formulation for our work.

We use the planning problem definition provided in [LaV06], which formally describes it as:

- A *state space* $S$, which is a finite and non-empty set of states.
- For each state $s \in S$, an *action space* $A(s)$.
- A *transition function* $f$, where $f(s, a) \in S$ for every $s \in S$ and $a \in A(s)$
- A *cost function* $c$, where $c(s, a) \in \mathbb{R}^*$ for every $s \in S$ and $a \in A(s)$
- An *initial state* $s_0$
- A *goal set* $S_G \subseteq S$

A plan $P$ in this formulation is a sequence of actions $P = \{a_0, a_1, ..., a_n\}$ such that, $\forall i \in [0, n]$, $a_i \in A(s_i)$ and $f(s_i, a_i) = s_{i+1}$. If $s_{n+1} \in S_G$ then the $P$ reaches a goal state and is considered a complete plan. If $P$ is such that $\sum_{i=0}^{n} c(s_i, a_i)$ is minimal, $P$ is an ideal plan. The goal of a planner is to find an ideal plan as fast as possible.

We can easily visualize a planning problem as a directed graph with weighted edges. The nodes of the graph are the states, there is an edge from state $x$ to $y$ if, and only if, there is an action $a \in A(x)$ such that $f(x, a) = y$ and the weight of that edge will be $c(x, a)$. A plan can be viewed as a path in the graph. The figure 2.1 illustrates this visualization.

#### Domain Example: Coin Flipper

We created a simple domain and problem to use as an example through the rest of this section. It will be useful to see how a simple example can become more complex as we relax the restrictions of the planning domain.

In our proposed example there is a row of $N$ coins, each showing either heads or tails, and the agent can flip any of the coins. We can consider that each possible configuration of coins is a state $s \in S$ and in every state there is an action that flips each of the coins. So, the domain for $N$ is:

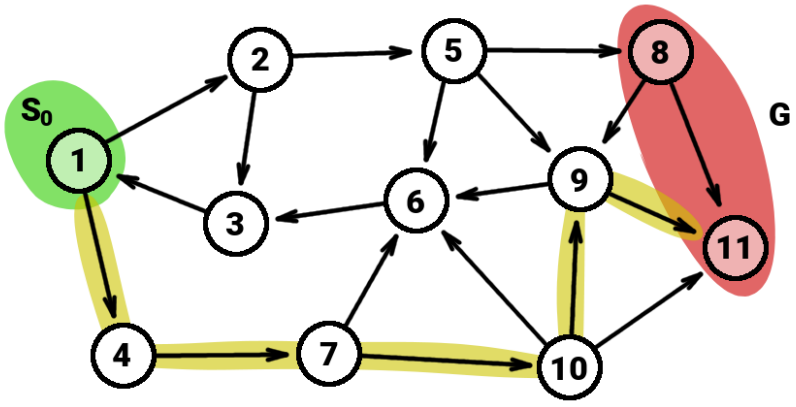


**Figure 2.1:** *Graph visualization of a planning problem. The initial state is highlighted in green, the goal set is in red and a plan in yellow. The edge weights have been omitted.*

- $S$ is the *state space* with $2^N$ states.
- For each $s \in S$ the *action space* $A(s)$ has $N$ actions.
- A *transition function* $f$ with $2^N N$ points in its domain.
- A *cost function* $c$ with $2^N N$ points in its domain. We can consider the value of $c$ to be constant if we wish to minimize the number of flips, regardless of coin.
- $s_0$ is some initial disposition of coins
- $S_G$ are all the acceptable dispositions of coins

It is easy to realize that the state space for even a simple problem such as this can quickly grow into problematic sizes. The exponential rate is due in great part to the fact that the planner knows nothing about the states and cannot simplify the problem's representation. This becomes clear if we create an instance of this problem with 2 coins:

- $S = \{s_1, s_2, s_3, s_4\}$
- $A(s_i) = \{a_1, a_2\}$, for $i \in [1, 4]$:
- $f$ such that:

$$f(s_1, a_1) = s_4;\ f(s_2, a_1) = s_3;\ f(s_3, a_1) = s_2;\ f(s_4, a_1) = s_1;$$
$$f(s_1, a_2) = s_2;\ f(s_2, a_2) = s_1;\ f(s_3, a_2) = s_4;\ f(s_4, a_2) = s_3;$$

- $c(s_i, a_j) = 1,\ \forall i, j$
- $s_0 = s_2$
- $S_G = \{s_1, s_4\}$

In this example the single action plan $P = \{a_1\}$ is optimal (as is $P' = \{a_2\}$). The interesting part of the example is to realize that a planner can find the solution even without any knowledge about coins or their flipping. It simply deals with states and actions. As the designers of the problem we know that $s_1 = \{heads, heads\}$, $s_2 = \{heads, tails\}$, $s_3 = \{tails, heads\}$ and $s_4 = \{tails, tails\}$. We also know that $a_1$ flips the first coin and $a_2$ flips the second.

### Solver Algorithms

A planning problem solver is essentially a search algorithm that can work in a generic, user-defined, domain. Many of the approaches used to solve planning problems are direct adaptation of traditional AI search methods. For a search algorithm to be used in planning it needs to ensure that a solution will be found if one exists. In finite state spaces this is easily ensured with a systematic search - one that explores every reachable state by keeping track of visited states and avoiding cycles. In this section we provide an overview of the main search methods developed in the early stages of planning research due to their importance as a stepping stone to improved search methods in newer planners. Most note worthy among the early search algorithms are:

- Breadth First ([RN09]): This search algorithm expands the search using a FIFO state queue (first in, first out). All the neighbors of the initial state (i.e., states reachable with a single action execution) are added to the queue and each step in the search removes the first state in the queue and adds its neighbors to the end of the queue (unless they were already visited). This causes the search expansion to occur in a uniform manner, with smaller plans always being encountered before bigger ones, hence the name breadth first.

- Depth First ([RN09]): This search algorithm expands the search using a stack (first in, last out). Every neighbor of the initial state is added to the stack and each step in the search removes the first state and piles all of its neighbors to the top of the stack (unless they were already visited). This causes the search to expand asymmetrically, investigating longer plans before evens starting on other, shorter, plans. The order in which plans are investigated is not well defined and it usually depends on the order the actions are defined in the domain specification.

- Iterative Deepening ([SA88], [RN09]): This search algorithm performs a depth first search with a limited depth and incrementally increases the depth to reach further and further. This search avoids potential shortcomings of the more greedy depth first while still managing to perform better than breadth first in the worse case scenarios.

- Dijkstra's Graph Search ([Dij59], [RN09]): The Dijkstra's search algorithm was originally developed while studying graph theory. It is useful in planning due to the direct way a planning problem can be represented by a graph. The main idea of the algorithm is that we can use a generic heuristic to choose which action to expand of first: the action cost. Instead of randomly exploring new states, Dijkstra chooses to always expand the plan with the lowest cost. The expansion of the search occurs differently depending on the available actions, but Dijkstra ensures that the first proper plan found is optimal. If the cost of every action is equal, this search will behave exactly like the breadth first search.

- A* Search ([HNR68], [RN09]): The A* is a widely used heuristic search method that was adapted for planning. The idea behind A* is to prioritize searching the most promising places first. In the case of planning the most promising plans, according to some heuristic, are expanded first. The heuristic will try to estimate the cost to reach a goal state from the current state. If we can ensure that this estimation is never greater than the actual cost of reaching a goal state, then it can be used to guide the search. If the heuristic is always zero, then A* degenerates to Dijkstra's graph search. It is worth noting that a heuristic is usually domain-specific and will remove the benefit of the planner being a general problem solver. In subsection 2.1.3 we will see that general heuristics can be created when we sophisticate our problem representation.

The search can start at the initial state and seek a goal, which is referred to as forward searching, or start a goal state and move backwards through the actions seeking the initial state, aptly named backwards searching ([RN09]). One can also seek in both directions at once, trying to meet in the middle, which is called a bidirectional search ([Poh69]).

**Advantages and Limitations**

The greatest advantage introduced by planning is the generic solver, the capacity of defining an input language and allowing the users to define both the domain (state and action space) and the problem (initial state and goal) in execution time. This property makes planning very useful when we consider AI design: the specialist whose knowledge we would like to incorporate into some AI system does not usually know how to express his knowledge in programming code. Planning provides a way for us to gather specialist knowledge and create automated systems without requiring specialists with very specific computer science capabilities. The simplicity of the graphical representation further facilitates the user's understanding of planning and how to input his knowledge into the system.

The most glaring limitation of this initial version of planning is the state space and action space sizes. As we hinted at when describing the coin flipper example in 2.1.2, the size of the representation can grow exponentially even in the simplest of domains. This is in great part due to the fact that the planner knows nothing about the problem it is solving. It cannot separate the coins in our example and instead must deal with the exponential number of possible coin configuration, for instance. This creates an issue for the users trying to specify the domain and for the planner trying to find solutions.

In subsection 2.1.3 we will see how a little more information about the domain allows planners to deal with larger state spaces while greatly reducing the domain representation size.

### 2.1.3 Logic-based Representation of Planning

The logic-based representation of planning problems uses boolean properties to describe the domain's states and actions. Instead of treating each state as a black box, this representation considers that states are constructed and that the building blocks are boolean values describing some property. If we consider our coin flipper example from 2.1.2 we can easily conclude that a single boolean property per coin is enough to describe any possible state in the domain. Each property can be true if the coin is showing heads and false otherwise, for instance. Unfortunately, not every domain can be so easily expressed and with such great gain (from $2^N$ states we consider $N$ variables), but several interesting classes of domains can. Formally the planning problem with a logic-based representation contains the following ([LaV06]):

- A nonempty set of *instances* $I$.
- A nonempty set of *predicates* $P$ where each $p \in P$ is a function $i \to [0, 1]$, for $i \subseteq I$. To put simply, a predicate assigns *true* or *false* to some subset of instances.
- A nonempty set of *operators* $O$, each with *preconditions* and *effects* and a cost $c \in \mathbb{R}^*$.
  - The *preconditions* are divided in positive and negative requirements, each being a set of predicates. For an operator to be applicable every predicate in the positive set must be true and every one in the negative must be false.
  - The *effects* are also divided in positive and negative, each being a set of predicates. When an operator is applied the predicates in the positive set become true and those on the negative set became false.
- An initial set $S_0$ which holds the predicates that must be true while all others are false.
- A goal set $G$ which holds the set of predicates that must be true and the set that must be false for the goal to be met.

The predicates in the above definition represent the boolean properties that can be used as building blocks of a state in the domain. Instances represent the objects in the domain that have properties that can be changed. In a domain where an agent can open a door, there would be an instance *door* and a predicate *IsOpen(door)* that can be true or false. We can consider the coin

flipper of 2.1.2 for a more concrete example of logic-based representation. The coin flipper problem in this representation would become:

- $I = \{c_1, c_2\}$
- $P = \{ShowingHeads(c_1), ShowingHeads(c_2)\}$
- $O = \{h_1, h_2, t_1, t_2\}$, where:

  $h_1$ requirements are: positive $= \emptyset$, negative $= \{ShowingHeads(c_1)\}$.
  $h_1$ effects are: positive $= \{ShowingHeads(c_1)\}$, negative $= \emptyset$.

  $h_2$ requirements are: positive $= \emptyset$, negative $= \{ShowingHeads(c_2)\}$.
  $h_2$ effects are: positive $= \{ShowingHeads(c_2)\}$, negative $= \emptyset$.

  $t_1$ requirements are: positive $= \{ShowingHeads(c_1)\}$, negative $= \emptyset$.
  $t_1$ effects are: positive $= \emptyset$, negative $= \{ShowingHeads(c_1)\}$.

  $t_2$ requirements are: positive $= \{ShowingHeads(c_2)\}$, negative $= \emptyset$.
  $t_2$ effects are: positive $= \emptyset$, negative $= \{ShowingHeads(c_2)\}$.

  The cost for every action is equal and one.

- $S_0$ predicates: $\{ShowingHeads(c_1)\}$.
- $G$ positive predicates: $\{ShowingHeads(c_2)\}$. And negative: $\{ShowingHeads(c_1)\}$.

An optimal plan to solve this problem would be $P = \{t_1, h_2\}$. The plan flips the first coin to show tails and then the second coin to show heads. One might note that we changed the goal from the original problem. Originally the goal of the problem was to have both coins showing the same face, which were states $s_1$ and $s_4$. With the proposed logic-based representation we cannot specify this exact goal, since $G$ only holds a set of positive and negative predicates. If we wanted to be able to specify such goal, we would need to create a predicate $ShowingSameFace(c_1, c_2)$ and increase the number of actions to comply with this added sophistication. We find this worth noting to illustrate that not all cases have a clean and simple logic-based representation.

### A Generic Heuristic

In 2.1.2 we reviewed the most note worthy methods of searching applicable to planning. The idea of a heuristic search was raised but it wasn't easy to find a general heuristic that could guide the search while maintaining the planner a general problem solver. The logic-based representation allows us to create some general heuristics due to the added information about state properties and how actions manipulate them.

A common practice to solve planning problems using the logic-based representation is to convert them to a satisfiability problem of propositional logic (SAT) and use a general SAT solver ([KS+92]). This solution allows us to make use of the considerable efforts that have been made in general SAT solvers. In [Rin12] a planning specific heuristic was added to the SAT solver in order to further improve performance.

Consider some predicate in the positive set of predicates for the goal $G$. If this predicate is not in $S_0$, then we know that there must be in the plan an action with that predicate in their positive effects. A valid heuristic can be created using this logic. For any state the heuristic will find all the actions needed to add the required positive and negative predicates in $G$, ignoring their relative order and requirements. If our action set has the minimum possible cost, then our heuristic value will never be greater than the actual cost. This method is covered in [McD96].

A planner using this representation can try to find the actions necessary to reach the goal and then redefine the goal as a state where those necessary actions can be applied, and so forth. [BF97] uses this representation to consider actions that can reach the goal, while initially ignoring action

interference. This allows them to create and use a small graph structure, when compared to the size of the state space, to search for a valid plan.

Of course this is simpler said than done, and in practice planners need to consider negative interactions and ordering. But the representation yields a considerable performance boost to planners without the requirement of domain specific heuristics.

#### Advantages and Limitations

A major advantage of logic-based representations is that we do not have to explicitly enumerate the whole state space. In fact, in many cases the planner will never have to visit most of the states in a domain. This simplification allows planners to deal with domain sizes that are considerably larger and introduces a new problem specification format that is more intuitive in many domains. Another advantage is the possibility of using the added information to create heuristics that improve on search performance.

As an example, the coin flipper that originally had $2^N$ states in its representation can now be described with only $N$ literals, $N$ predicates and $2N$ actions. Some work has been done to allow actions to define anonymous requirements and effects, which further reduces the size of the representation. For instance, a *flipToTails* action would require that *some* coin was showing heads and the result would reference that *same* coin, saying it no longer showed heads.

It is worth remembering that not all domains can be neatly expressed with logic-based representation, as we have seen in the goal definition of our coin example. Since we only work with boolean properties, many domains still cannot be modeled properly. We cannot represent quantities (e.g., amount of fuel on a rocket, available money, etc) or precise positioning (i.e., we can say whether or not the agent is in some room, but cannot give his world coordinates).

Up until this point, every action is deterministic: the result of executing an action predictably takes the agent from one state to another. While this is enough for a great number of interesting domains (e.g., chess, Go), we require some way to model the uncertainty of actions in other domains (e.g., basketball, auto piloting drones, blackjack). In subsection 2.1.4 we will see how this problem was tackled by the planning community and some important concepts we use in this thesis.

### 2.1.4 Probabilistic Planning

An important expansion of planning representation power was the possibility to model non-deterministic actions. In this new probabilistic environment an action taken at some stage can lead to any number of different states, according to some probability distribution. The rest of the problem specification remains the same, but the solver can no longer simply provide a plan due to the uncertainty of the actions. A solver in a probabilistic environment needs to provide a policy, which is a map that provides an action for each relevant state in the domain. A relevant state is one that can conceivably be reached from the initial state if one follows the provided policy at every step. States that cannot be reached are omitted from the policy for performance improvements. [AK01] observes that a policy can be viewed as a finite state machine (discussed here in 2.2.1), which provides an interesting visualization mechanism for the resulting behavior of following some policy.

A probabilistic planning problem modeled in this manner is called a Markov Decision Problem (MDP) and it has been greatly covered in literature ([Put14], [Alt99]). The idea of using a policy instead of a simple plan allows the agent to recover from execution problems and provides some graceful degradation on well modeled domains. The planning problem also shifted from a goal seeking problem to a reward maximization problem: instead of defining goal states a problem sets rewards for each state and the planner should find a policy that maximizes the reward over some finite (or infinite) amount of time.

Formally, we can defined a MDP as follows:

- A *state space* $S$, which is a finite set of states.

- For each state $s \in S$, an *action space* $A(s)$.
- A *transition function* $p$, where $p(s'|s,a)$ is the probability that executing action $a$ in $s$ will lead to $s'$, for $s, s' \in S$ and $a \in A(s)$.
- A reward function $r$, where $r(s,a)$ is the reward of executing action $a$ in state $s$, for $s \in S$ and $a \in A(s)$.
- An initial state $S_0 \in S$.

The use of rewards turns the problem from a seeking problem to a maximization problem. It is interesting to note that we can still represent problems that deal with action cost and goal states using this framework. Suppose we want some set $S_G \in S$ to be goal states and we have some cost for every action. We can set the costs as a negative rewards and create zero-cost idle actions $i$ such that $i \in A(s)$ if, and only if, $s \in S_G$ and $p(s|s,i) = 1$. The ideal policy will be to reach a goal state while minimizing the negative rewards (costs) and then execute the idle action indefinitely. In any case, the planner will seek a solution in the form of a policy $\pi$ that maps an action to every reachable state. A reachable state is defined as any state that can possibly be reached from $S_0$ if $\pi$ is followed at every step.

As an example of this, we can consider the coin flipper example from 2.1.2. If instead of simply flipping a coin the agent needs to toss it, then we can model the problem as a probabilistic planning problem. Each action would toss a different coin and it could lead to a new state or remain in the same state with equal probability (assuming balanced coins). The problem could be modeled as follows:

- $S = \{s_1, s_2, s_3, s_4\}$
- $A(s_i) = \{a_1, a_2\}$, for $i \in \{2, 3\}$.
- $A(s_i) = \{a_1, a_2, idle\}$, for $i \in \{1, 4\}$.
- $p$ such that:

  $p(s_4|s_1,a_1) = 0.5$; $p(s_3|s_2,a_1) = 0.5$; $p(s_2|s_3,a_1) = 0.5$; $p(s_1|s_4,a_1) = 0.5$;
  $p(s_2|s_1,a_2) = 0.5$; $p(s_1|s_2,a_2) = 0.5$; $p(s_4|s_3,a_2) = 0.5$; $p(s_3|s_4,a_2) = 0.5$;
  $p(s_i|s_i,a_j) = 0.5$ for $i \in [1,4]$ and $j \in [1,2]$.
  $p(s_i|s_i,idle) = 1$ for $i \in \{1,4\}$.

- $r$ such that:

  $r(s_i|s_i,a_j) = -1$ for $i \in [1,4]$ and $j \in [1,2]$.
  $r(s_i|s_i,idle) = 0$ for $i \in \{1,4\}$.

- $s_0 = s_2$

A possible policy for this problem would be $\pi(s_1) = idle, \pi(s_2) = a_2$. States $s_3$ and $s_4$ are not reachable from $s_2$ following $\pi$, so they can be omitted. Following this policy the agent would stay in $s_2$ flipping the second coin until if comes up heads, from then on it would just take the *idle* action indefinitely.

This formalization of probabilistic planning does not use the logic-based representation, but many works in literature have proposed ways to introduce this optimization to probabilistic planning. The important distinctions between the enumeration and the logic-based representations have already been covered in subsection 2.1.3. This document will not elaborate further on the probabilistic logic-based representation, except to note that it has been throughly covered in planning literature ([BDG00], [KP00], [GKPV03]). The performance impacts of switching the representation of probabilistic planners are studied in [LGM98].

### Solver Algorithms

A solver in probabilistic planning can no longer be called a general domain search, as was the case for deterministic planning. Since a solver needs to develop a policy, the entire reachable domain must be studied and ideal actions must be calculated. The literature has produced many interesting algorithms to tackle this problem, some of interest are:

- Value Iteration ([Bel13]): The planner creates a function that estimates the reward an agent would have in some fixed amount of time should he perform an action on some state. This function is calculated for every state and action on each iteration step. The initial value is simply the reward expectancy of applying some action on a state, trivially calculated. Then every step the calculation increases the fixed amount of time and updates the values based on previous results. The calculation will eventually converges to a precise reward expectancy for an infinite horizon. The policy is extracted by simply choosing the action in each state that maximizes the calculated function. This method updates the whole policy at every step, regardless of the likelihood of an agent finding themselves in some particular state during policy execution.

- Policy Iteration ([KP00], [RN09]): In policy iteration the planner starts with some random policy. Then, in each step of the iteration the planner evaluates the current policy and tries to improve it. This algorithm updates the entire policy every step by analyzing in each state if a different action would improve the expected reward. As with the value iteration, the calculation eventually converges and an optimal policy can be extracted.

- Real Time Dynamic Programming ([BBS95]): The planner works in a similar manner to value iteration, but the states are updated asymmetrically. In every step of the iteration the planner simulates the execution of some fixed amount of decisions using the policy and updates only states reached during this simulation, usually called a trial. Each trial chooses the current best action and randomizes the next state based on the transition function, repeating this for a limited number of transitions. The idea is that more frequently visited states will update faster than hard to reach states. The unreachable states are never updated, creating a performance improvement. There is an issue with the convergence speed due to some states being updated infrequently. A solution is encountered by [BG03], where the problem is corrected by labeling states that have already converged to avoid revisiting them.

### Advantages and Limitations

Probabilistic planning using MDPs allows for the modeling of environments with uncertainty and it expands considerably the domains that can be solved by planners. We are able to maintain the general nature of the planners, still allowing the problems to be specified independently from the solvers. Another advantage that is maintained is the intuitive nature of the graphical representation of a problem and of policies. The only difference from the original representation is that edges of the graph will split into multiple states, each branch with their own probability. In figure 2.2 we can see an example of this representation.

Probabilistic planning requires that the exact probabilities that govern the actions of the domain be completely know and precise. This is not always the case, and, furthermore, we may not even know all the possible outcomes of an action execution in some domains. When considering sports (among many other domains), we might have an estimation of the success rate of an action (e.g., the chance that a player might score a free throw), but the other results are unknown (e.g., if the player misses a free throw, the ball's resulting position is unpredictable). The issue of continuous domains is still not handled by this approach.

It is worth mentioning that probabilistic planning has been used extensively in the RoboCup competitions, described in [KTS+98]. While there are several similarities between our work and what is created for the competitions, the main focus shifts from a strategic point of view to a more

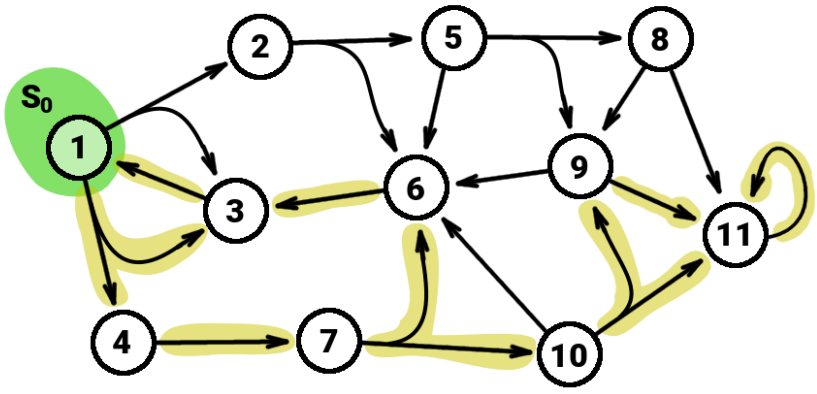


**Figure 2.2:** *Graph representation of a MDP. $S_0$ is highlighted in green and a policy is highlighted in yellow. The rewards have been omitted and the policy has not been calculated for the unreachable states.*

physical one. Solutions need to consider sensors and actuators (e.g., [LSR05]), requirering early implementation of partially observable environments, as done by [PTCd05]. Our approach allows us to consider only the collective planning initially, without the burden of sensor planning and physical controllers.

## 2.2 Behavior Simulation

In this chapter we will review the main behavior simulation techniques we used as the starting point of our individual planner and mechanical animation control. Finite state machines, covered in subsection 2.2.1 were once used as the go-to technique to implement behavior in simulations and games. Nowadays behavior implementation is done with newer techniques, but we still see the use of finite state machines in animation control. Subsection 2.2.2 talks about behavior trees, an improvement over state machines that is more commonly used today.

### 2.2.1 Finite State Machines

Finite state machines (FSMs) are a modular and incremental way of implementing agent behavior. They were very popular in simulator and game development, specially to implement computer controlled adversaries, such as enemy agents in competition games ([MF16]). FSMs were very present in literature and many improvements were developed, such as hierarchical FSMs, but nowadays their use is usually limited to simpler animation state control, which we will cover in subsection 2.3.2.

A behavior simulating FSM can be represented by a graph where each node is a simpler behavior than the one being simulated and each transition is an environment condition. For instance, we could consider a simple FSM to control an enemy agent in an adventure game. The states of this machine would be "Patrol", where the enemy follows some predetermined route, "Pursue", where the enemy approaches the player and "Attack", where the enemy hits the player. The initial state would be "Patrol" and transitions would be linked to the player. If the agent has visual contact with the player he transitions to "Pursue" and if the distance to the player becomes smaller than his reach he transitions to "Attack". The figure 2.3 illustrates this FSM. While each state contains only a simple and easily implemented behavior, when combined using the FSM's structure these gain complexity.

An interesting trait of FSMs is that they allow for incremental development of the agent's behavior. In our enemy example above, a developer might start by implementing a very simple version of each behavior. He could then sophisticate the behavior of each state incrementally,

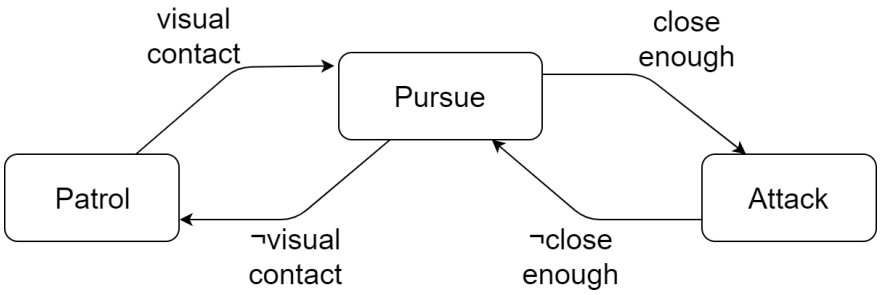


**Figure 2.3:** *A simple FSM that controls an enemy agent capable of patrolling the environment, pursuing the player and attacking.*

implementing obstacle avoidance in the "Pursue" state, for instance. This trait allowed FSM to be used by large developing teams by decreasing the overhead associated with multiple programmers working on the same task.

Since a FSM creates a more sophisticated behavior from simpler ones, a natural idea is to create another layer of sophistication by using FSMs as states of more complex FSMs. This idea became what is now known as hierarchical finite state machines and there is abundant literature on the subject. The idea of composing simple behaviors into more complex ones shows up again in literature when we study behavior trees, covered in subsection 2.2.2.

A FSM can facilitate the design of agent behavior by non-specialist, since the structure can be assembled without deep knowledge of the underlining code guiding the behavior nodes. While this is an immense leap forward when compared to simply coding the whole behavior, FSMs are still a direct code implementation with improved organization. New behaviors still need to be implemented in code before they can be used as the building block in a FSM. This highlights one of the problems with FSMs: they are domain specific. New domains, or even new actions for an existing domain, require new implementation.

### 2.2.2 Behavior Trees

Behavior trees (BTs) implements complex behaviors by creating a tree where the leaves are simpler behaviors and the branches perform selections and compositions. BTs were initially developed in the video-game industry to control the enemy's AI ([Isl05]). Since then, they became and remain quite popular both in video-game AI as well as other applications ([LBC10], [GLPH15], [CMDO16]). The Mario AI ([KT12]), a benchmark used to test video-game artificial intelligence has been used by many BT researchers as a testing ground for their work ([PNOB11]).

When simulating an agent's behavior, the system will begin at the root of the BT and follow the adequate branches until reaching a leaf node, which it will then execute. The behavior defined in a leaf node can accept parameters, which further expands the usability of the BT ([SGJ+11]). As with FSMs, BTs have a natural sophistication with hierarchical behavior trees, where a single leaf node is substituted by a whole BT.

Navigation through the BT is dependent of the result of the execution of the leaf nodes. Each execution can return with one of three possible results: *success*, *failure* and *running*. Success signals that the behavior executed completely and correctly, failure means something went wrong and running means the behavior has not yet completed its execution. A running result means that in the next iteration the BT can simply continue that execution without further calculations. A leaf node that implements movement might return running while moving the agent to the target location, success when the agent reaches the location and failure if something happens that impedes the movement (e.g., a doorway closes, fuel ends).

The branch nodes are compositions of behavior capable of decisions. There are two main types of composition nodes, the AND and the OR. An AND composition node will try to sequentially execute each of their child nodes and return success if, and only if, every child returns success. If any of the children nodes return executing, the AND node will return executing as well and

continue execution of that child in the next iteration. If any of the children returns failure, the AND node will return failure as well. This type of node is useful when creating complex behavior where each part depends on the last. For instance, an AND node to "deal with hunger" might have three children: "find food", "pick food" and "eat food".

The OR composition node will execute each child until the first execution returns a success. If a child returns failure, the OR node will move onto the next. Any success will cause the OR to return a success as well. If no children return success, the OR will return a failure. If a child returns executing the OR node will also return executing and go back to that child in the next iteration. The OR node is useful for complex behaviors that have many ways to achieve the same result. For instance, an OR node to "leave house" might have two children: "leave through door", "leave through window".

The creation of BT is done by generating simpler trees and composing them together with AND or OR nodes. One of the main advantages of this technique is that error handling becomes an integral part of the system, not an exception to be handled. This and the ease of composition make BTs more appropriate then FSMs to control a group of agents, as highlighted by [CO14]. If some behavior fails, for whatever reason, the tree will simply advance the execution to the next state and carry on from there. A feature kept from FSMs is that the behavior can be designed and visualized by non-programmers, as done in [GLPH15]. A drawback also kept from FSMs is that BT are domain specific. If the agent needs to be able to execute a new action, a new implementation will be needed.

## 2.3 Mechanical Simulation

We reefer to mechanical simulation as the work of properly placing agent's limbs to reflect the properties of the physical simulation. For instance, if the agent is moving at a walking speed we expect the agent's limbs to reflect a walking motion, if he moves faster, we will expect a running motion to be represented. In simulations and games that create realistic 3D models, this is done though the manipulation of preexisting animations. An animation is a sequence of frames, each frame containing the positions and rotations for each limb of the agent. For instance, a 30 frame animation might represent a walk cycle by creating a sequence of frames where the agent steps forward twice, making sure the first and last are compatible so the animation can play in a loop and the agent will appear to be walking forward. This chapter will cover how this animations are acquired, composed and edited to create the complex motions required to simulate the mechanical movement of the agent.

### 2.3.1 Animation Acquisition

The animation of a 3D character's model is done through the use of a skeleton which manipulates the vertexes' positions. In order to better understand that statement we should review the components of a 3D model and of a skeleton. A 3D model is a way of representing the surface of some object in three dimensional space (e.g., a character, a table, a ball). The most important elements of a 3D model are:

- Vertexes: A vertex is simply a three dimensional coordinate that is used to define the faces of a model.
- Faces: A face is a polygon created from some of the vertexes of the model. The faces are the only visible part of a model and they define the object's surface.

It is worth mentioning that other elements exists (e.g., texture coordinates) but they are not important to understand model animation. In order to change the position of a model, one simply needs to change the position of the model's vertexes. Moving vertexes individually is not practical and usually not necessary. In a consistent model, such as a human being, many vertexes can be

grouped and moved together, the vertexes defining the skull, for instance. For this reason the animation skeleton was created. A skeleton is simply a group of bones, where each bone contains:

- Pivot point: A point three dimensional space that represents the root of the bone. This is the point around which the bone will be rotated.
- Direction: The direction the bone is facing at resting position.

In order to use a skeleton to animate a 3D model the bones need to know which vertexes they should manipulate. This assignment of vertexes to bones is done though a process called skinning. Skinning creates groups of vertexes and assign those groups to bones, for instance every vertex defining the skull might be assigned to a bone called "head". If a bone is rotated, every vertex in its group will be moved in a way that maintains relative position to the bone's pivot and direction in the resting position. While this should be enough to understand skeleton animation, it is important to note that many improvements have been added to this system over time, such as:

- Allowing a vertex to be influenced by more than one bone. This is useful in vertexes near joints, allowing a smooth transition instead of an unnatural break in the model.
- Allowing a bone to be scaled, therefore inflating or deflating the vertexes in its groups
- Creating bone rotation constraints that restrict unnatural bone rotations.

When the skinning process is completed a pose can be created by setting the rotation of each bone. An animation is a sequence of poses, each called a frame, that represent the desired movement. An animation can be created by specialized software or by recording the movement of real people ([RRP12], [OKA11]).

### 2.3.2 Animation Blending

Animation clips can be created to represent many different motions an agent might need to execute, but it is difficult to have a single clip for every possible variation and combination of movements available to the agent. For instance, we might have a walk, a jog and a run animation, but the agent can move in any speed between a light walk and a hard run. To solve this problem animation blending was introduced ([KG03]). An animation blend takes the base clips and some parameter and defines a transition function to be executed. In our movement example the walk, jog and run clips would be the base and the movement speed would be the transition parameter. An ideal movement speed would be assigned to the clips (e.g., walk is $5km/h$, jog is $10km/h$ and run is $13km/h$) and any speed in between these values would use a weighted blend of the clips in either end of the interval (e.g., a movement of $7km/h$ would be 60% walk and 40% jog). A blend is a weighted geometric average of the bone rotations of every pose, so good results require that the clips be compatible, for instance, every move cycle should start with the same foot going forward in our example.

It is also possible to create masks in order to blend different animation clips to different parts of the body. For instance, we might want the lower body to execute a movement animation while the upper body holds a ball instead of swinging the arms to the side as usual. This result can be achieved without the need of a new set of movement animations. It is done by defining a body mask to isolate the upper body bones and, for those bones only, blending an animation with the arms holding a ball.

Which animation clip (or blend) is being used at any given time is usually controlled by a FSM. The states are animations and transitions are control parameters, such as movement velocity or boolean values to control jumping, ball possession, tiredness, etc. The support for body masks is done via layered FSM, where each layer has a weight and a mask assigned to it and the states of every layer are combined using these weights and masks into a final motion. The use of FSMs to control animations allows for very complex movements to be defined without the need of any type of coding, which allows for a quick creation, test and improvement cycle.

### 2.3.3 Inverse Kinematics

Some cases require a fine adjustment of the animation with awareness of the simulation environment. The hand placement to pick up objects is an example of this, when an agent is standing in front of a desk and wishes to pick some object on top of it, for instance. It would be very difficult to keep a list of animation clips to lead the agent's hand to every conceivable spot on the table, so a generic animation that can be adjusted to the situation is preferred. This adjustment is done through a process called inverse kinematics (IK), covered in detail by [Wel93]. The control is done over a chain of bones where the pivot point of a bone depends on the rotation of the previous bone in the chain (e.g., upper arm → lower arm → wrist). Target point and rotation are defined and the IK system tries to find a rotation for each bone in the chain that brings the last bone's position and rotation to the defined target. This is often done to correctly place the feet of walking humanoids, as described by [GM85]. The hybrid control system where a preexisting blend of animations is adjusted by some real time calculations is covered by [SPF03].

A system of rotation constraints helps to keep the IK calculations realistic and a body mask ensures that the adjustments do not interfere with other parts of the body. IK is used by animation software to facilitate the posing process, as it is often easier to move the extremities (e.g., hands and feet) than to specify every joint rotation. In addition, IK is also applied during execution of simulations to provide fine animation tweaking, ensuring that the characters feet always touch the ground or his hands properly grab items, for instance.

# Chapter 3

# Formal Strategy Model

We developed a formal strategy model capable of describing a control system for a team of agents in complex environments. While our initial effort was aimed at using the strategy model to describe team strategy in invasion team sports environments (e.g., soccer, basketball, hokey), the resulting system is generic enough to be used outside this domain. In its essence, the strategy model creates equivalence classes of the domain's state space and thus allows for a compact representation of only the strategically significant data. This chapter will present the definitions for each component of the formal model in section 3.1 and then proceed to show how we created a formal basketball strategy model from this definitions in section 3.2.

## 3.1 Formal Definitions

The formal strategy model creates a structured representation of the desired collective behavior of a team of agents collaborating towards some shared goal. It is a description language for any team strategy. We call a specific team strategy described using our model a *Team Strategy Model* (TSM).

Our model was created to control a team of agents in a cooperative dynamic stochastic system. The system can also be competitive, in which case each team will use their own TSM to control their actions. A domain is compatible with our formal model if it has:

- A *state space* $S$, which can be infinite and non-enumerable, but where each state $s \in S$ can be described by a finite set of variables $V$.
- An *action space* $A(s)$ for each state $s \in S$.
- A non-empty set of *possible outcomes* $O$, $\forall a \in A(s)$, $\forall s \in S$. An outcome $o \in O(s, a)$ is a state of the state space $S$.
- For each $s, s' \in S$ and variable $v \in V$ such that only the value of $v$ differs between $s$ and $s'$, there is an outcome $o \in O(s, a)$ for some action $a \in A(s)$ such that $o = s'$.

Note that both the state space and the action space can be infinite. The first due to continuous variables in $V$ and the second by utilizing parametric actions (e.g., an action that changes the value of a variable by some constant). Any domain can be compatible with our model if: 1 - states that can be described by a finite set of variables; 2 - we know at least one possible outcome for every action; 3 - we have a control action that can change the value of each variable. However, not every compatible domain will be a good fit for our model. Since we aim to faithfully implement some strategy, domains where the goal is to achieve some objective regardless of strategy and only optimizing cost are ill suited for our approach. The best domain fits we found so far are those where one wishes to simulate human behavior or gather specialist knowledge, such as sport simulations.

A TSM can be applied to domains where the state space is infinite due to the use of equivalence classes to restrict and group the state space to contain only relevant strategic situations. Each

relevant equivalence class in the TSM is stored in a structure called the strategy map and the actions required to transition between two strategy maps are stored in a structured called the strategy transition. The composition of these two structures creates the strategy graph, where some important strategy navigation information of the TSM is encoded. The remaining of this section will define each of these structures and also explain how a TSM can be used to analyze real world events.

### 3.1.1 Strategy Map

The strategy map (SM) is a representation of a class of relevant situations for some strategy defined in a TSM. The SM can group any number of states in the domain's state space into a single state in the TSM. Our formal model allows the SM to create equivalence classes in one of two ways: information omission or information grouping.

Information omission is analogous to the "don't care" variables used when creating intervals in boolean algebra ([HP68]). If a state can be described by a set of variables $V$ and there is a subset $V_s \subseteq V$, then an equivalence class can be created by assigning values to every $v \in V_s$ while omitting all $v' \notin V_s$. Every state in the domain for which the values of every $v \in V_s$ coincide with the assigned values will be captured by this equivalence class, regardless of the values of the variables not in $V_s$. One could create an all encompassing class by setting $V_s = \emptyset$, which can be useful in error handling.

Information grouping defines discreet and finite intervals to describe continuous variables in a manner analogous to interval creation in continuous variables statistical analysis. The interval creation is done by delimiting the possible values of some continuous variable. Take for instance some $v \in V$ such that $v$ is a real number ($v \in \mathbb{R}$), then we can define values $a, b \in \mathbb{R}$ and the interval $[a, b]$. Every state where $v \in [a, b]$ could be captured by the defined equivalence class. More broadly, any variable in $V$ that belongs to a group with well defined order (e.g., $\mathbb{N}$, $\mathbb{I}$, $\mathbb{R}$) can be described by an equivalence class though this method.

For each variable in a domain, the SM can either create an interval or omit the variable completely. The SM also defines a weight for each variable, which signifies the relative importance of that constraint over the others. This weight is useful when measuring how closely a SM is describing some domain situation. We will see it used in chapter 4.

For a domain with set of variables $V$, a SM is formally defined as a set of constraints $C$, where each $c \in C$ has:

- An *associated variable* $v \in V$, such that $v \in G$ and $G$ is an ordered set with order $<_G$.
- A *start* and an *end* bounds $a, b \in G$ such that $a <_G b$.
- A *weight* $w \in \mathbb{R}^*$.

Figure 3.1 illustrates the process of creating SMs in a domain with two continuous variables ($v_1$ and $v_2$). Both maps define a constrain for each domain variable. $SM_1$ constraints the variable $v_1$ with $c_1$ and the variable $v_2$ with $c_2$. $SM_2$ constraints the variable $v_1$ with $c'_1$ and the variable $v_2$ with $c'_2$. It is worth noting that not every possible combination of values for $v_1$ and $v_2$ is covered by the SMs. This is not required because the SMs only need to cover value combinations that are relevant for TSM. Chapter 4 will introduce a method to work with value combinations not covered by the TSM.

### 3.1.2 Strategy Transition

A strategy transition (ST) contains the necessary information to inform the system of how the transitions between SMs should occur in the TSM. A ST defines the actions that should be executed when transitioning between states. It would be impossible to specify valid actions to transition between every possible combination of the original domain states that are represented in the SMs, since the number of combinations can be infinite and domain actions do not apply

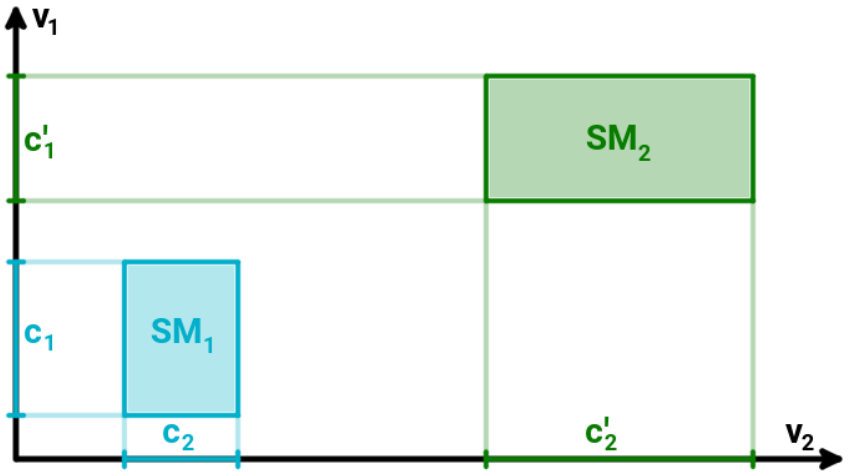


**Figure 3.1:** *Two strategy maps ($SM_1$ and $SM_2$) defined in a continuous domain with two variables.*

to classes of states. What the ST does instead is specify domain actions necessary to transition between two representative states of the original domain, one of each SM. While the defined actions might not accomplish the transition between all the combinations, it provides enough information to guide the transition using the methods described in chapter 4. We define a ST as follows:

- An origin SM $\alpha$.
- A destination SM $\beta$.
- Representative states $\alpha_r \in \alpha$ and $\beta_r \in \beta$.
- A list of actions $\{a_1, a_2, ..., a_n\}$ with $a_i \in A(s_i)$ and $s_i \in S$, $\forall i \in [1, n]$ such that:

  There is list of states $\{s_1, s_2, ..., s_{n+1}\}$ with $s_j \in S$, $\forall j \in [1, n+1]$ where:
  $s_{x+1} \in O(s_x, a_x)$, $a_x \in A(s_x)$, $s_1 = \alpha_r$ and $s_{n+1} = \beta_r$, $\forall x \in [1, n]$

Note that we use the notation $\alpha_r \in \alpha$ to indicate that $\alpha_r$ is a state in the original domain that obeys every constraint defined in $\alpha$ (i.e., $\alpha_r$ belongs to the equivalence class defined by $\alpha$). So, the ST selects a representative for each map and provides a list of actions that *may* take the system from one representative to the other. A list is valid as long as one of the possible outcomes of every action leads to the next step in the chain that eventually leads to the chosen destination. Figure 3.2 illustrates how a ST might be defined to transition between $SM_1$ and $SM_2$.

It is important to note that $\{a_1, a_2, ..., a_n\}$ are not necessarily the most efficient way to transition between $\alpha_r$ and $\beta_r$. They simply represent what the designer of the strategy would like the agents to perform when they find themselves in $SM_1$ and wish to reach $SM_2$. The choice of actions, representatives and even SMs are up to the designer and our objective is to apply them faithfully, not necessarily to control the agents in the best possible manner.

### 3.1.3 Strategy Graph

The strategy graph is a natural structure to represent the TSM: a directed graph with SMs as the nodes and STs as the edges. We use the graph to encode extra navigational information about the TSM. On each edge of the graph (i.e., for each ST) we encode a risk value that represents the probability of something going wrong during the transition and on each node (i.e., each SM) we encode a reward value representing how advantageous we consider the situations described by that SM to be. Usually, a planning problem will work with only one of those variables, but adapting the algorithm to work with both values is trivial and we found that such representation is more easily understood by the users.

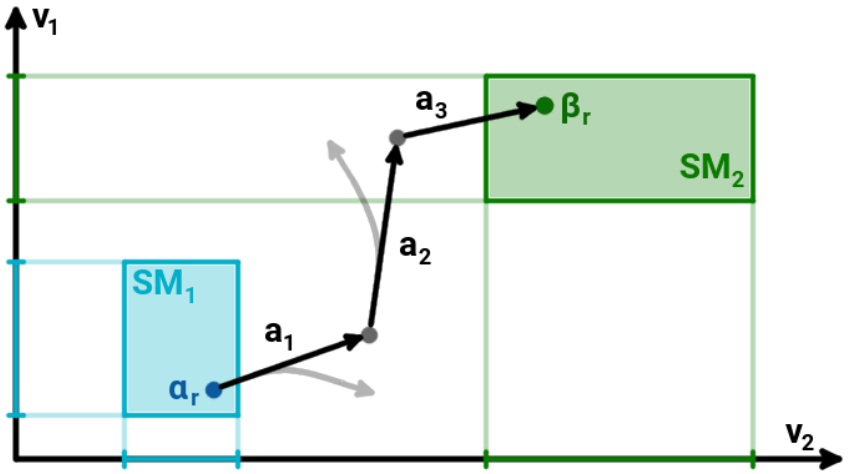


**Figure 3.2:** *A strategy transition with actions* $\{a_1, a_2, a_3\}$ *and representative states* $\alpha_r \in SM_1$ *and* $\beta_r \in SM_2$*. Other outcomes of actions* $a_1$ *and* $a_2$ *have been faded to improve readability.*

If we know the probability distribution of the actions in the domain, we have a good candidate for the risk value of a ST: one minus the product of the probability of each action leading to the next state in the chain. This is not required, the risk and reward values encoded in the strategy graph can be provided by the user or learned by the system through simulations.

Considering the risk to be the probability of a transition to fail and the reward to be the expected accumulated reward from that state after some time, we can estimate those values after several simulation rounds.

We define a strategy graph as:

- A set of states $S$, each $s \in S$ contains:
  - A SM $m$.
  - A reward value $\gamma \in \mathbb{R}^*$.
- A set of edges $E$, each $e \in E$ contains:
  - A ST $t$.
  - An origin state $\alpha \in S$.
  - A destination state $\beta \in S$.
  - A risk value $\omega \in [0, 1]$.

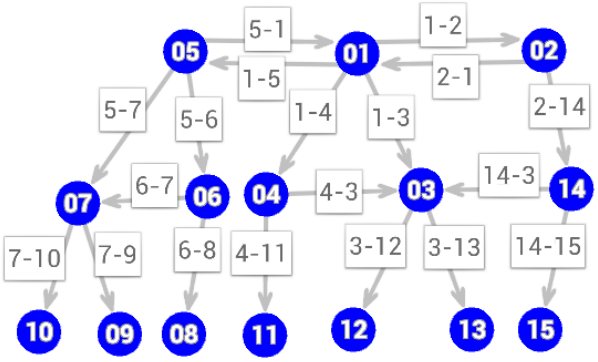


**Figure 3.3:** *A simple strategy graph, the values for the risks and rewards have been omitted.*

Figure 3.3 shows what a simple strategy graph might look like. The amount of states in the graph and the number of edges leaving each state are indicatives of the complexity of the strategy encoded within. As we will discuss further on, our tests indicated that fairly complex behavior emerges with only a few dozen states in the strategy graph.

A graph contains the whole TSM of a team, but we do not need to use the whole model all the time. It is possible to define subsets of the graph to be used at different times, for instance, one could exclude some states and transitions that represent riskier behavior in order to apply a safer strategy. A FSM can be defined to control which subset of the graph should be active at any given moment: the states of the machine would be graph subsets and the transitions would be changes in key control parameters. Figure 3.4 illustrates such a FSM. This sophistication allows the TSM to dynamically change the team's behavior based on high level changes in the environment.

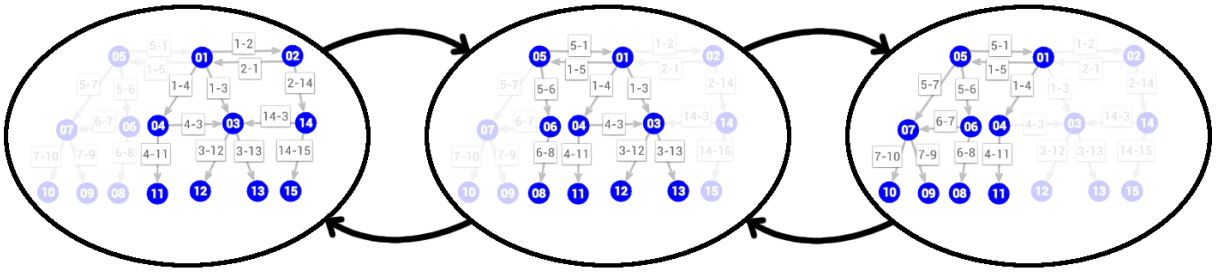

**Figure 3.4:** *A FSM with 3 states, each containing a different subset of a strategy graph.*

### 3.1.4 An equivalent representation with individual agents

So far, our model relies on what we call a collective brain: we assume that a single entity holds the strategic knowledge and any planning using such model will be done by this entity and then passed on to each agent. While this approach works very well for many cases, there is an increased level of realism that could be achieved by switching to an autonomous agents approach. The main idea is to execute the collective planning in the context of a single agent and then synchronize the collective actions through a communication layer. The agent executing the planning is decided by some domain rule, for instance, the player with the ball in a basketball domain. Figure 3.5 illustrates this representation. Note how each agent has his own version of the TSM and a communication layer is necessary to achieve the collective behavior calculated by one of the agents.

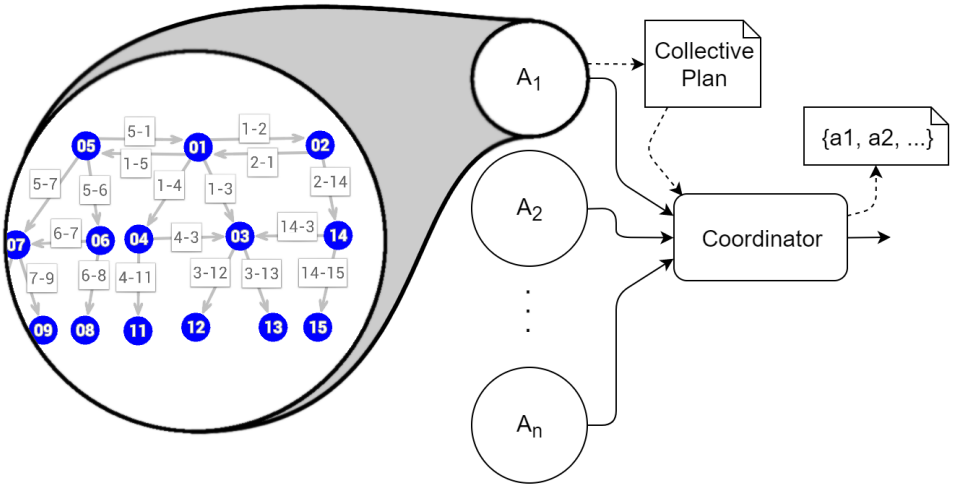


**Figure 3.5:** *The equivalent representation of the formal model using individual agents.*

If the TSM version is identical for every agent and the communication is both immediate and

flawless, then both representations will be exactly the same. However, this autonomous agents approach allows the addition of a new level of sophistication. We could generate more complex behavior by implementing slightly different TSM for each agent. We could also introduce some error in the communication layer to simulate real world difficulties. Finally, this autonomous agent approach could allow us to create a partially observable domain where each agent has a different perception of the situation and acts accordingly. This added level of control has not been fully explored by us yet, but it is suggested as a future work in section 7.

There have been many recent works in literature that deal with coordination of many agents in a system ([QMSW16]) and how to balance between long-term and short-term objectives ([NPLOB16]), such as following the collective plan and dealing with individual needs. Task delegation between agents has also been covered in literature ([CF98], [FC01]).

### 3.1.5 Event analysis using the formal model

A direct result of creating a formal strategy model is that we can use it to analyze real world events that take place in the domain described by our model. For any domain where we can capture real world events and infer their corresponding state in the domain's state space, we can use a TSM to measure how well the event matches the TSM. We have so far used this technique in the domain of invasion team sports ([LSOB14]), where a state in the domain can be usually described by the position of all the players and the ball. Some studies have also attempted to infer the strategy from real world events ([LBC+12]).

There are significant efforts in the sport scientific community to capture player's movements and positioning during matches. This information can be used to detect how closely a team is following some TSM defined for that domain. We can extract the frequency that each play is executed and how often the team deviates from the strategy (i.e., how often the disposition of the team doesn't match any state or transition in the TSM). We can also detect possible problems in the application of the strategy, if the team uses a play with lower success probability more often then another play with better results, for instance. Once we have the TSM and a digital representation of the event, we can extract the following:

- Frequency of plays: How often each sequence of STs (i.e., play) is executed.
- Risks during the event: We can calculate the risk for every ST executed during the event. This can even be used to improve the value in the TSM with a large enough number of occurrences.
- Rewards during event: Just like the Risk, we can calculate the rewards acquired for each state visited during the event.
- Strategy deviation: We can detect how often the team deviated from the TSM, and by how much if we use the metric defined in section 4.1.1.
- Strategy application: We can measure how well the plays were chosen by the team using the play score metric defined in 4.1.2. How well the team applied the strategy can be calculated by comparing the distribution of the plays executed and their ideal distribution, defined by the score metric.

This analysis can also be done over digital simulations of matches, of course. This eliminates the considerable problem of acquiring the positioning of the players in a real match. Another way to deal with this acquisition difficulty is to work with much less information. Some studies limit themselves to analyze the game just around the ball ([GBD97], [BJM02], [SD06]) or just for one of the teams ([GCG10]). Our TSM works with equivalence classes so we can define very broad classes and use only approximate positioning information to analyze the match. We have worked with goalkeeper analysis where the match information required could be extracted by watching a regular transmission of the game. The only information needed was the area in the field where the

player with the ball was located and the goalkeeper's general position. Of course the amount of interesting information that can be extracted increases with more precise data, but we managed to compare the performance of goalkeepers in several match situations from only this small scrap of data.

## 3.2 Basketball's Strategy Model

We developed a formal strategy model for basketball with the goal of representing strategic knowledge in a structured manner that was useful for both match analysis and simulation by computer systems. This thesis focuses mainly in the simulation of matches using this model, but we have successfully used the same techniques in match analysis of both basketball and soccer.

Our research of sport literature into strategy models did not yield studies that had the depth and formalism that we required. We found studies whose purpose was to investigate recurring behavior patterns in matches and their relation to the success of the team ([LJS+11], [BJM02], [GBD97], [GCG10], [SD06], [THRB10]). The analysis conducted in these studies, in general, do not describe the events in a formal manner and there is a lack of precision in the description that impacts the behavior analysis. On top of a possible loss of relevant information, this lack of precision prevents the extraction of strategy models that we can use in match simulation. Match analysis in literature is often limited to a single team, or just around the ball, due to limits in the recording and annotation of matches. This further complicates the creation of a unified model for the complete description of the strategy of both teams.

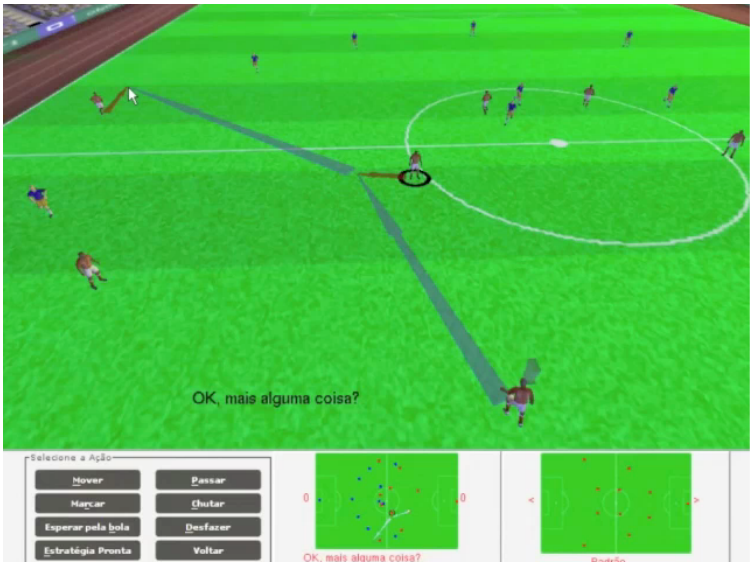


**Figure 3.6:** *The user interface for Strategy Soccer.*

We began our efforts to create a strategy model with an experiment modeling the behavior of support players in soccer. The idea of this first experiment, called *Strategy Soccer*, was to make the user define the intelligent actions around the ball and try to automatically move the rest of the players according to some strategy model. This very simplified model yielded promising results: a reasonably complex team behavior was simulated when the user actively engaged the system by providing key actions to players around the ball. The collective control was created using some user provided maps that had the ideal positions of every player in the team. We allowed the user to create any number of maps, called "plays", before the match began and then he could activate a play as a way of indirectly controlling players. Any player that was not directly controlled by the user would follow the active play, moving to his designated position. In figure 3.6 we can see the user interface for Strategy Soccer. Note that the user is assigning a few actions to the players (indicated by arrows on the field) and he can switch the active map for each team on the bottom

menu. The simulation is paused as he does this and will resulme once he is done. Complex team behavior emerged when the user provided a large amount of plays and frequently toggled the active play. While still depending heavily on the user, the results made us reason that we could simulate complex behaviors if we managed to automatically toggle the plays and somehow include ball handling actions in the model.

We shifted to basketball as a simulation domain due to the availability of complex strategies in basketball literature. Many well documented styles of basketball strategy painted a picture of complex behaviors filled with subtleties, even though the descriptions lacked the formal rigor we required. Our reasoning is that, if we can simulate basketball, we will be able to shift to simpler domains without major difficulties.

The creation of the formal model described in section 3.1 was an evolution of the Strategy Soccer trial that included the means to describe automatic play control and complex player actions. In the basketball model a map describes a class of situations that could be pursued by the players and a transition describes the type of actions needed to go from one class of situations to another. In the remaining of this section we will describe how we instantiated the formal definitions of SM, ST and the strategy graph to be applied in the basketball domain.

### 3.2.1 The Basketball's Strategy Map

The SM in basketball is a representation of a class of match situations idealized in the TSM defined by the user. In basketball a match situation can be defined by the positioning of each player and the position of the ball. We defined the variables in $V$ for this domain as:

- A *position* in $\mathbb{R}^2$ for each player.
- A *velocity* in $\mathbb{R}^2$ for each player.
- A *rotation* in $\mathbb{R}$ for each player.
- A *ball holder index* in $\mathbb{N}$.
- A *ball position* in $\mathbb{R}^2$.
- A *ball velocity* in $\mathbb{R}^2$.

The SM creates what we call a *player role* for each match player in order to create the necessary equivalence classes using the concept of "playing area". The playing area is an elliptical interval we used to group the position and rotation information into an equivalence class. Furthermore, a SM does not need to specify every player role, omitted ones will simply enlarge the defined equivalence class to encompass any position and rotation for the omitted player. The player role can also indicate whether or not it has the ball. Every other variable in the domain is grouped by omission (i.e., ignored by the SM). Figure 3.7 shows an example of a SM where every player role was defined and player role 1 has the ball. Only the playing areas for the offensive team have been highlighted for simplicity.

Formally, we define a SM as a group of *player roles* $P$, where each player role $p \in P$ has:

- A team identifier $t \in \{\text{Offense, Defense}\}$.
- A two dimensional *center* $(x, y) \in \mathbb{R}^2$
- A *rotation* of the playing area $\alpha \in \mathbb{R}^2$. This is the angle between the positive x-axis and the facing of the playing area, counter-clock wise.
- A *rotation tolerance* $\alpha_t \in \mathbb{R}^*$.
- A *dimension* of the playing area $(w, h) \in \mathbb{R}^2$.
- A boolean value indicating *ball possession*.

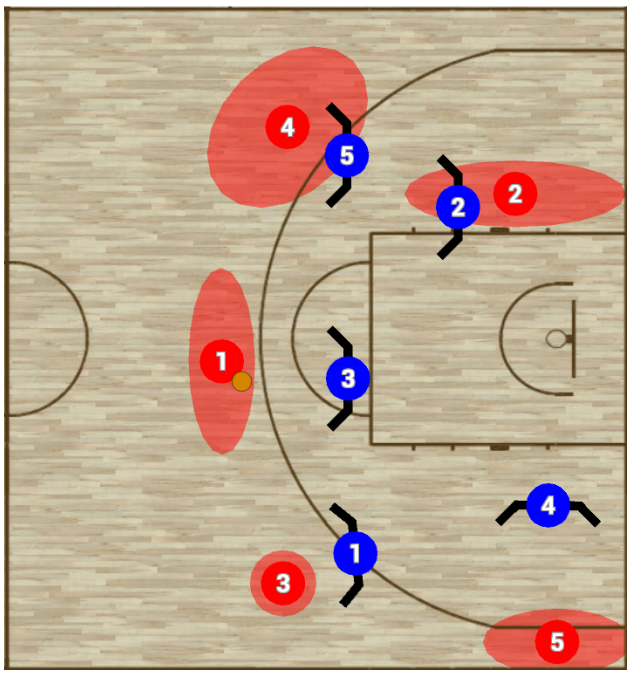


**Figure 3.7:** *Strategy map with the playing areas highlighted in red for the player roles in the offensive team.*

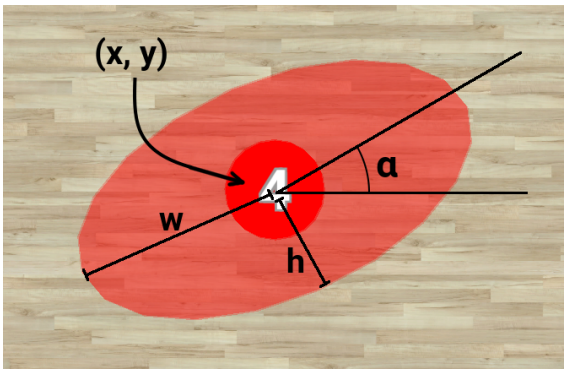


**Figure 3.8:** *Player role in a strategy map with position (x, y), rotation angle α and dimension (w, h) of the playing area.*

It is not easy to create a set of SMs that cover every possible match situation, much less one that makes sense strategically. Fortunately such a set is not required in our proposed method. We created a matching function that allows any situation to be assigned to an existing SM, even if some error is introduced. This method is covered in detail in section 4.1.1, but for now it means that the specialist describing the SMs needs to consider only those that are important to the TSM he is creating. The number of SM necessary to describe complex strategies is quite small. Some of the traditional basketball strategies (e.g., triangles offensive), can be described with no more than a few dozen SMs. This small number of SMs greatly facilitates the handling of TSMs by computer systems, as well as facilitating the visualization by humans.

### 3.2.2 The Basketball's Strategy Transition

A ST allows the specialist to detail the necessary actions the players must execute in order to transition between SMs, these include: path and velocity of movement, passes, throws, idle wait times, cuts, etc. The timing of the actions (e.g., when the path of two player roles will cross) can be specified through a combination of movement velocity and idle wait times.

In order to choose the representative states for each SM, we consider the position of every player to be the center of the player role and we disregard the rotation tolerance. Any omitted player

roles are put into predetermined positions outside the court. This SM representatives place each player in the mean position of every match situation encompassed by the SM. They also provide a natural representation of the transition when overlaid with the maps, since the movement starts and ends in the center of the playing areas.

We define a ST as follows:

- An origin SM $\alpha$.
- A destination SM $\beta$.
- A set of paths $P$, one for each player role in $\alpha$ and $\beta$.
- Each path $p \in P$ contains a nonempty list of waypoints $W(p)$, each waypoint $w \in W(p)$ has:
  - An initial wait time $t \in \mathbb{R}^*$.
  - A start position $(x, y) \in \mathbb{R}^2$.
  - A movement speed $s \in \mathbb{R}^*$.
  - An end position $(x', y') \in \mathbb{R}^2$.
  - A final wait time $t' \in \mathbb{R}^*$.
- A set of passes $B$, each $b \in B$ has:
  - A start time $t \in \mathbb{R}^*$.
  - The path $p \in P$ that initiates the pass.
  - The path $r \in P$ that receives the pass.
- An optional shoot to basket time $t \in \mathbb{R}^*$.
- Obeying the following restrictions:

A valid transition has a few requirements that need to be enforced when it is being created. These ensure that the ST contains an applicable set of actions whose execution can lead from $\alpha$ to $\beta$:

1. The start position of the first waypoint in each path must coincide with the position of the corresponding player role's center in $\alpha$.
2. The end position of the last waypoint in each path must coincide with the position of the corresponding player role's center in $\beta$.
3. Each waypoint after the first on a path must have a start position equal to the previous waypoint's end position.
4. Each waypoint except the last on a path must have an end position equal to the next waypoint's start position.
5. For every pass the player role that initiates the pass must have ball possession at the start time of the pass (either by starting with the ball or being the receiver of a previous pass).
6. The player with ball possession after all passes have been accounted must be the player with the ball in $\beta$.

Figure 3.9 shows an example of a ST for basketball. The continuous lines represent player movement and the segmented line represents a pass. A continuous wavy line is overlaid on the movement of the player that has the ball and the T shaped arrowhead represents a screen action executed by player 3. These overlays can be calculated automatically because the information contained in the definition of a ST is enough for us to extract the ideal position of every player during the transition. We can use this information to detect actions with special semantics in basketball, such as the screen performed by player 3 or the carry action executed by players 5 and then 1. The time duration of the transition can also be calculated easily with the available data and therefore it does not need to be specified.

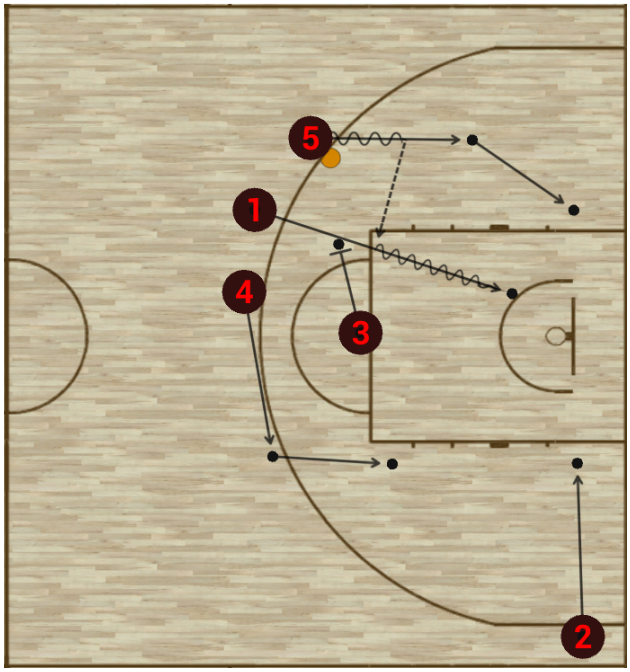


**Figure 3.9:** *A strategy transition with a single pass from player 5 to player 1. The paths of players 5 and 4 have 3 waypoints while the rest have only 2.*

### 3.2.3 The Basketball's Strategy Graph

For the basketball's strategy graph we chose to interpret the risk as the probability of a transition failing due to some action not yielding the desired result. This means a pass failing due to an interception by the adversary or a fumble or a carry action failing due to the ball being stolen. The reward of a state is simply the probability of the team scoring some points after being in that state. Both these values can be specified by the user and they could also be estimated though simulation after several rounds, as suggested in the future works section (7).

We define the strategy graph for basketball as:

- A set of states $S$, each $s \in S$ contains:
  - A state id.
  - A SM $m$.
  - A reward value $\gamma \in [0, 1]$.
- A set of edges $E$, each $e \in E$ contains:
  - A transition id.
  - A ST $t$.
  - An origin state $\alpha \in S$.
  - A destination state $\beta \in S$.
  - A risk value $\omega \in [0, 1]$.

The structure of a graph carries some information about the TSM it represents. A large number of central nodes, each with several branches leading to terminal nodes, indicates a strategy with plenty of options on how to better exploit the adversary's defense. Terminal nodes (i.e., nodes without exiting edges) are states with score attempts, and a large number of those is indicative of a more unpredictable strategy. If we take as an example the graph in figure 3.3, we can see that a team can transition between the states in the top row until an opportunity arises and a play towards a terminal state is started. While we have not yet studied these properties in depth, we

believe there is much to be learned from studding TSM's graph structures. Such study is suggested and explained in further detail in our future works chapter (7).

A FSM to control which subset of the strategy graph is active during any time in the match is very useful to enhance the adaptability of the strategy. Each subset of the graph can hold only the appropriate plays for some match situation. In basketball the match situations that could be used to trigger a state change in the FSM are:

- Amount of time left on the clock
- Match score
- Players on the court
- Fouls and penalties awarded to players
- Current quarter being played

With these situations a designer could create a specific set of plays that should only be executed if a certain player is on the court, or a set to be executed only if the team is winning by a certain amount of points. The possibilities are endless and this extra layer of control allows the designer to adapt his strategies to every conceivable scenario.

## 3.3 Basketball Defense's Reactive Behavior

In this section we will show how we managed to create a simpler strategic model to control the reactive behavior of the defensive team in a basketball match. During the development of the basketball's formal model we realized that the behavior of the defense can be conditioned by what the offensive team is doing. This realization led us to create a dedicated and simplified model that was better suited to control the defense. In [BA98] a similar approach is proposed to control a team of autonomous robots moving in formation using reactive behavior. We call the model reactive because, during a match, the offense dictates the evolution of the play while the defense simply reacts trying to minimize the opportunities given to the offensive players until the ball possession can be taken. This means that the ideal positioning for the defense is conditioned by the position of the offense with this approach.

A collective defense behavior is defined by ideal position and rotation of every defense player for any distribution of the offense. So the challenge of this model is to be able to identify equivalent offensive distribution and specify ideal positioning to react to each of them. The allocation problem, where one defender is allocated to each attacker, is not considered here. How the allocation is accomplished will be better explained in 4.1.1. The rest of this section will describe the simplified strategy model we created for the defense and how we used it in our system.

### 3.3.1 Basketball Defense's Strategy Model

A defensive player behavior in basketball depends on two factors: the position of his target for marking and the position of the ball. Our model will define equivalence classes for the positioning of these two players in the court and use these classes to list every strategically significant situation for the defense. To accomplish this we divided the court into 14 areas of interest, 10 of them being two mirrored pairs of 5 zones. The court division creates the following areas:

- Home (HOME): The home court for the offense.
- Top Key (TK): Consider a *central triangle* to be a triangle with a point below the basket, a base on the central court line and edges that intersect the paint corners. The part of the central triangle that is outside the 3-pts line is the Top Key.

- Short Top Key (STK): The part of the central triangle (defined above) that is inside the 3-pts line but outside the paint.
- Paint - Central (P(C)): The part of the central triangle (defined above) that is inside the paint.
- Paint - Left (P(L)): The left half of the remainder of the paint once the central triangle is removed.
- Corner - Left (C(L)): The rectangle on the left side of the court, outside the 3-pts line, from the end of the court until the bend in the 3-pts line.
- Short Corner - Left (SC(L)): The rectangle on the left side of the court, inside the 3-pts line and outside the pain, from end of court to the bend in the 3-pts line.
- Wing - Left (W(L)): The area between the Top Key and the Corner - Left, outside the 3-pts line.
- Short Wing - Left (SW(L)): Area between the Short Top Key and the Short Corner - Left.
- Paint - Right (P(R)), Corner - Right (C(R)), Short Corner - Right (SC(R)), Wing - Right (W(R)) and Short Wing - Right (SW(R)): These areas are mirror images of their left side counterparts.

Figure 3.10 contains a diagram of the court division.

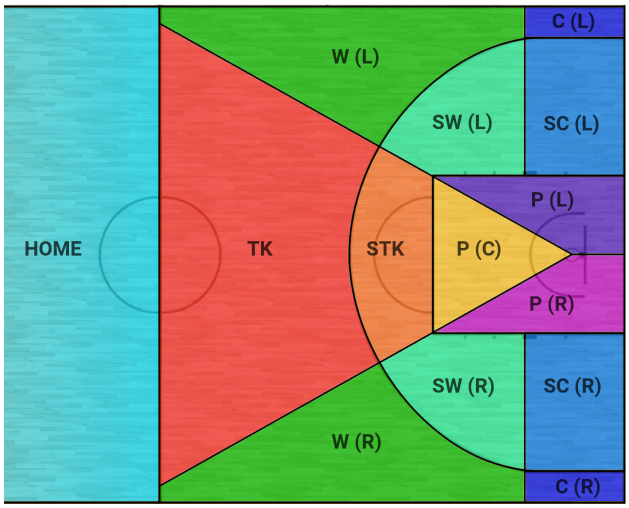


**Figure 3.10:** *Court division used in the definition of the equivalence classes in the TSM of the basketball's defense.*

A marker needs to know the area where his target is located, the area where the ball holder is located and whether or not his target is the ball holder. This is all the necessary information to calculate a reasonable position and rotation for the marker. We also make the reasonable assumption that mirror situations can be dealt with in the same manner, effectively halving our state space. So a SM in this model is simply a pair of areas in the defined court division. And since the ideal behavior of a marker is simply a position and rotation, the ST in this model is simply a move action towards some objective.

We created a set of actions that can define the movement objective using as reference the position of the target, the position of the ball and some court features (e.g., basket position, end line, side line). The idea is to allow the behavior to be easily defined in a generic manner for any situation, therefore reducing the amount of work for the strategy designer. A generic description of a movement objective would be "stay between the ball holder and the basket, 1m away from the

ball holder, looking at the ball", for instance. The three variables defined in a movement objective are:

1. *Direction*: A line segment over which the marker should be. The possible values of *direction* are:
   - Cover Basket: The line connecting the target and the basket.
   - Cover Ball: The line connecting the target and the ball holder.
   - Cover Center: The line connecting the target and the center of the court.
   - Cover Paint: The line between the target and the paint line.
   - Cover 3-pts: The line between the target and the 3-pts line.
2. *Distance*: A distance between the marker and one of the points defining the direction line segment. If the direction is "cover paint" or "cover 3-pts line", we use the distance from marker to line, otherwise we use the distance from marker to target.
3. *Gaze*: The point where the marker should direct his gaze. Possible values of *gaze* are:
   - Look at target: The marker looks directly to the target.
   - Look at ball: The marker looks directly to the ball holder.
   - Look at sideline: The marker looks to the nearest sideline.
   - Look at end line: The marker looks to the opposite end line.
   - Look between ball and target: The marker looks to the point halfway from the target to the ball, trying to see both with peripheral vision.

The strategy graph in this simplified model can be omitted because the transitions will happen naturally during simulation and each plan for the defense will be comprised of a single ST. This is due to the reactive nature of the defense, the actual navigation through the strategy graph is controlled by the offensive team, not through choice of the defense.

# Chapter 4

# Planning with the Strategy Model

The complex domain defined in 3.1 can be translated into a much simpler planning domain with the use of the TSM, as chapter 3 has shown. In this chapter we will show how we can apply plans calculated in the TSM to some state in the original domain. For the remainder of this chapter, we will call a state in the original domain a *situation*. Section 4.1 will define the 3 planning steps formally and section 4.2 will show how we implemented them in the basketball domain from section 3.2.

## 4.1 Formal Definitions

The objective of the planning process is: given some initial situation, calculate applicable actions that are consistent with some input strategy. One issue we need to deal with is that the situations and actions exist in a different domain than the one where the strategy is defined. In order to solve this we created 2 translation functions, one that maps a situation to a state in the strategy and a second that maps transitions in the strategy to actions in the original domain. Our approach to planning using a TSM requires 3 separate steps:

1. Matching: Translate a situation into a SM of the TSM.
2. Planning: Extract a plan from the TSM.
3. Realization: Adapt the plan so it fits the situation.

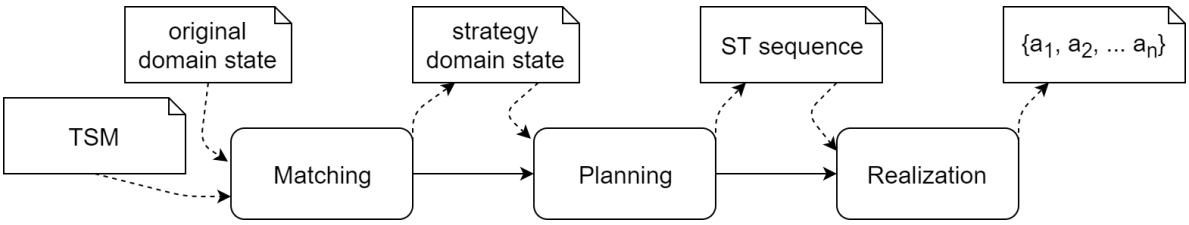


**Figure 4.1:** *The planning process, comprised of matching, TSM planning and realization.*

Figure 4.1 illustrates the 3 steps in the process. The dotted lines indicate data transmission and the continuous line indicates process order. If we consider the whole process as a black box, the input is a situation and a TSM and the output is an action sequence. The remainder of this section will describe in detail how each of the 3 steps can be accomplished.

### 4.1.1 The Matching Function

The objective of the matching function is to find the SM in the TSM that best represents some situation. In order to do this, a metric is established that measures how well any given SM represents a situation. While it would be trivial to search for a SM whose equivalence class encompasses the

situation, we have no guarantees that such a SM exists for every situation in the original domain. A metric solves this problem by allowing some error to be introduced to the matching process when required. This is similar to the nearest-neighbor approach in machine learning. As noted in [BL97], such approach is very sensitive to what attributes are used in the metric. We follow their idea of defining weights for each variable used by the metric, thus providing a tool to remove irrelevant attributes.

Given a metric $\mu$ to measure distance between SMs and a given situation, the matching function is quite simple: simply return the SM in the TSM that minimizes the distance to the situation being matched. In order to define $\mu$, consider the following:

- Let $\{v_1, v_2, ..., v_n\}$ be the set of all domain variables in $V$.
- Let $\{G_1, G_2, ..., G_n\}$ be groups with orders $\{<_{G1}, <_{G2}, ..., <_{Gn}\}$ such that $v_i \in G_i$, $\forall i \in [1, n]$.
- For each $i \in [1, n]$, let functions $\delta_i$: $G_i^2 \to \mathbb{R}^*$ be such that $\delta_i(x, y)$ with $x, y \in G_i$ represent the distance between $x$ and $y$ in $G_i$.
- Let a situation be described by the variables $\{x_1, x_2, ..., x_n\}$.
- Let a SM be defined by:
  - A set of omitted variables $O = \{o_1, o_2, ..., o_k\}$.
  - A set of grouping intervals $I = \{q_1, q_2, ..., q_l\}$, with components $q_i.a$ and $q_i.b$ being the start and end bounds respectively.
  - Such that $k + l = n$.
- Without loss of generality assume $SM = \{O, I\} = \{c_1, c_2, ..., c_n\}$ and that $c_i$ is the constraint of $x_i$, $\forall i \in [1, n]$.
- Let $\{w_1, w_2, ..., w_n\} \in \mathbb{R}^n$ be the weights of the constraints of SM.

With the above assumptions we are ready to define the metric $\mu$ to compare the situation described by $\{x_1, x_2, ..., x_n\}$ and a SM. We do this by creating sub-metrics $\{\mu_1, \mu_2, ..., \mu_n\}$ such that:

$$\mu_i = \begin{cases} 0 & \text{if } i \leq k \text{ or } q_i.a <_{Gi} x_i <_{Gi} q_i.b \\ min(\delta_i(x_i, q_i.a), \delta_i(x_i, q_i.b)) & \text{otherwise} \end{cases} \tag{4.1}$$

Then we can finally define $\mu$ simply as:

$$\mu = \sum_{i=1}^{n} \mu_i w_i \tag{4.2}$$

For a variable $v \in G$, the distance function $\delta : G^2 \to \mathbb{R}^*$ can be any domain specific function that suits a need, but it is worth mentioning that the Euclidean distance works well when $G$ is $\mathbb{I}^2$ or $\mathbb{R}^2$. When $G$ is $\mathbb{I}$ or $\mathbb{R}$ the absolute value of the difference also works well. In any of the two cases, one might want to consider using the squared value of the result in order to penalize greater differences in favor of multiple smaller ones.

Figure 4.2 illustrates the matching function. Simply put, the matching function will calculate a distance between each variable in the situation's description and the corresponding constraint in the SMs. Then it calculates a weighted sum using those distances and selects the SM that minimizes the sum to represent the situation in the TSM.

### 4.1.2 The Planning Function

The objective of the planning is to calculate a sequence of STs that starts in the SM found by the matching function. The risk and reward values in the strategy graph are used to calculate the

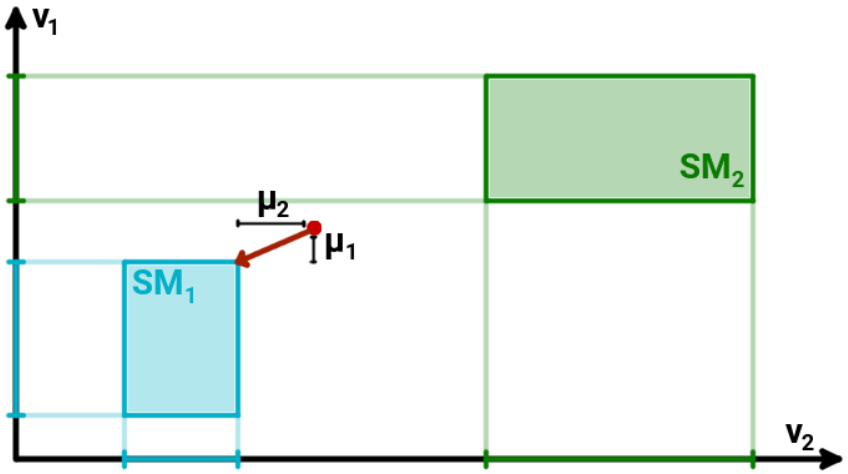


**Figure 4.2:** *An example of the matching function. The situation marked in red is matched to $SM_1$ after $\mu_1$ and $\mu 2$ are calculated.*

ideal sequence of STs, which we will refer to as a *play*. So a play is a path in the strategy graph that begins in the SM that best describes the current domain situation, as calculated by matching. It would be trivial for this process to calculate a play that maximizes some constraint, such as reward divided by risk, specially due to the small size of the strategy graph. However, the objective is to extract different plays in a manner consistent with the TSM, resulting in a more realistic and unpredictable behavior.

We would like to explore the TSM in a manner that is consistent with the intent of the strategy designer. Therefore, we cannot always simply pick the play that maximizes some constraint, but we can use the constraint to prioritize our play selection. We do this by defining a *score* for each play and then we use this score to define a distribution over which we will randomly select the play we want. Ideally, the plays with lower risk and greater rewards in the TSM should be picked more often, while increased risks or lower rewards should decrease the chance of a play being selected. This is a balance between exploration (trying every play) and exploitation (using better plays more often). This approach borrows heavily from the literature on Monte Carlo Tree Search methods ([BPW+12]). These methods have been successfully used in domains where there is uncertainty or the domain is not fully observable ([Cou07], [JGT14], [GWMT06], [CBSS08]).

Let's define a score function $\psi$ for some play defined in a strategy graph with states $S$ and edges $E$:

- Consider a play as $p = \{e_1, e_2, ..., e_n\}$, with $e_i \in E$, $\forall i \in [1, n]$ and:
  - $e_i.\beta \in S$ is the destination state of edge $e_i$.
  - $e_i.\beta.\gamma \in \mathbb{R}^*$ is the reward value of state $e_i.\beta$.
  - $e_i.\omega \in \mathbb{R}^*$ is the risk value of edge $e_i$.

Then $\psi$ is a function $\psi{:}E^n \rightarrow \mathbb{R}^*$ calculated as:

$$\psi(p) = \sum_{i=1}^{n} (1 - e_i.\omega)\ e_i.\beta.\gamma \tag{4.3}$$

We use $(1 - e_i.\omega)$ because it has an intuitive meaning: if $e_i.\omega$ is the probability that a transition will fail, then $(1 - e_i.\omega)$ is the probability that it will succeed. The function $\psi$ manages to capture the reward associated with a play in the strategy graph as well as the probability of acquiring the reward. If the reward associated with any state is zero, or if the risk associated with any edge is one, then the value of $\psi$ is zero. This might not be desirable, since it would mean the state (or

edge) is useless in the strategy and so is any play using it. To avoid this we can introduce a small value $\delta \in \mathbb{R}^+$, $\delta \ll 1$ and use it to clamp the values in the calculations, like so:

$$\psi(p) = \sum_{i=1}^{n} max(\delta, (1 - e_i.\omega))\ max(\delta, e_i.\beta.\gamma) \tag{4.4}$$

This ensures that every play as a non-zero probability of being picked, which makes sense if we consider that: 1 - the play can be extracted from the strategy graph; 2 - the user designed the strategy graph to contain the play. We consider two methods of picking a play, the long-term and the short-term planning:

- Short-term planning: A play is assembled one edge at a time, starting at the initial state and adding edges until there are no more eligible edges. An eligible edge is an edge whose origin state is the current state and whose destination state is not yet part of the play. To pick the next edge we list every eligible edge and pick one as follows:
  - Let $\{e_1, e_2, ..., e_n\}$ be the set of eligible edges in $E$.
  - Let $\{\psi(\{e_1\}), \psi(\{e_2\}), ..., \psi(\{e_n\})\}$ be the score of the single edge plays defined above.
  - Let $T = \sum_{i=1}^{n} \psi(\{e_i\})$.
  - Pick an edge in a way that $P(e_i) = \psi(\{e_i\})/T$ is the probability of edge $e_i$ being picked, $\forall i \in [1, n]$.
- Long-term planning: Every possible play that starts from the initial state is listed and one is selected as follows:
  - Let $\{p_1, p_2, ..., p_k\}$ be the set of possible plays.
  - Let $\{\psi(p_1), \psi(p_2), ..., \psi(p_k)\}$ be the scores of those plays.
  - Let $T = \sum_{i=1}^{k} \psi(p_i)$.
  - Pick a play in a way that $P(p_i) = \psi(p_i)/T$ is the probability of play $p_i$ being picked, $\forall i \in [1, k]$.

A simple way to pick a play such that each $p_i$ has the probability $P(p_i) = \psi(p_i)/T$ of being chosen is to uniformly pick a value $v \in (0, T]$ and choose the play $p_i$ such that:

$$\sum_{j=1}^{i} \psi(p_j) < v \leq \sum_{j=1}^{i+1} \psi(p_j) \tag{4.5}$$

An analogous method can be used to select the next edge in the short-term play calculation. This method is illustrated in figure 4.3, where one can note that the order in which the plays are arranged does not interfere in the fairness of the selection. The algorithm to accomplish this is described in more detail in chapter 6.

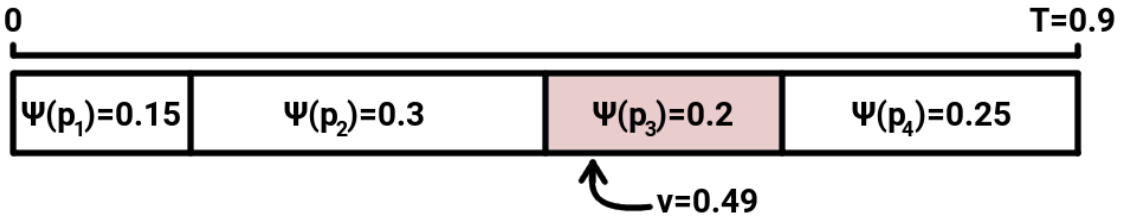


**Figure 4.3:** *The selection of play $p_3$ after the value $v$ was uniformly selected in the interval $[0, T]$.*

The difference between the methods of planning are indicative of the behavior we want to simulate. The short-term approach utilizes a greedy algorithm to create a play, which means that the immediate edges that are more attractive get selected more often. This could lead down a path with less attractive options, resulting in an overall worse play being selected more often than better options. This is not necessarily a bad thing, in some domains the greediness of this algorithm and the resulting quirk in the selection could be desirable traits for the simulation. If that is not the case, the long-term approach calculates an overall ideal distribution by comparing full plays. The difference in performance from both approaches can be ignored due to the small size of the strategy graphs.

To illustrate this process, consider the simple strategy graph in figure 4.4. If the matching algorithm finds $S_0$ as a starting point, there are 3 possible plays to be executed: $p_1 : S_0 \to S_1 \to S_3$, $p_2 : S_0 \to S_1 \to S_4$ and $p_3 : S_0 \to S_2 \to S_5$. This is how the short-term approach will select which play to execute:

- Initially, the short-term approach will calculate the score of the single edge plays that can be executed from $S_0$ and randomly select one:

$$\psi(\{S_0 \to S_1\}) = (1 - 0.5) * 0.6 = 0.3$$
$$\psi(\{S_0 \to S_2\}) = (1 - 0.3) * 0.7 = 0.49$$
$$T_1 = 0.3 + 0.49 = 0.79$$
$$P(\{S_0 \to S_1\}) = 0.3/0.79 = 0.38 = 38\%$$
$$P(\{S_0 \to S_2\}) = 0.49/0.79 = 0.62 = 62\%$$

- From there, the next plays will be scored and picked in a similar manner. If $\{S_0 \to S_1\}$ is chosen in the first step, then:

$$\psi(\{S_1 \to S_3\}) = (1 - 0.4) * 0.8 = 0.48$$
$$\psi(\{S_1 \to S_4\}) = (1 - 0.3) * 0.5 = 0.35$$
$$T_2 = 0.48 + 0.35 = 0.83$$
$$P(\{S_1 \to S_3\}) = 0.48/0.83 = 0.58 = 58\%$$
$$P(\{S_1 \to S_4\}) = 0.35/0.83 = 0.42 = 42\%$$

- Else, if the first step chose $\{S_0 \to S_2\}$, then the only option will be $\{S_2 \to S_5\}$ and it will be chosen with 100% probability.

- So, we can calculate the probability of each complete play being chosen by this approach:

$$P(p_1) = P(\{S_0 \to S_1\}) * P(\{S_1 \to S_3\}) = 0.38 * 0.58 = 22\%$$
$$P(p_2) = P(\{S_0 \to S_1\}) * P(\{S_1 \to S_4\}) = 0.38 * 0.42 = 16\%$$
$$P(p_3) = P(\{S_0 \to S_2\}) * P(\{S_2 \to S_5\}) = 0.62 * 1 = 62\%$$

This example illustrates the potential flaw in the greedy approach: $p_3$ is chosen more often even though it ends with a very high risk edge. This happens because, by the time this edge rolls around, we are already committed to that play. This does not happen in the long-term approach, where the play to be executed is selected as follows:

- Each complete play is scored and randomly selected:

$$\psi(p_1) = (1 - 0.5) * 0.6 * (1 - 0.4) * 0.8 = 0.14$$
$$\psi(p_2) = (1 - 0.5) * 0.6 * (1 - 0.3) * 0.5 = 0.10$$
$$\psi(p_3) = (1 - 0.3) * 0.7 * (1 - 0.9) * 0.7 = 0.03$$
$$T = 0.14 + 0.10 + 0.03 = 0.27$$

$$P(p_1) = 0.14/0.27 = 52\%$$
$$P(p_2) = 0.10/0.27 = 37\%$$
$$P(p_3) = 0.03/0.27 = 11\%$$

Now we can clearly see that $p_1$ is a more attractive overall option (according to $\psi$), and the long-term approach avoids the pitfall in $p_3$, only picking it 11% of the time. Again, a greedy agent might in fact fall into the pitfall, so the short-term approach is valid, depending on the type of behavior we intend to simulate.

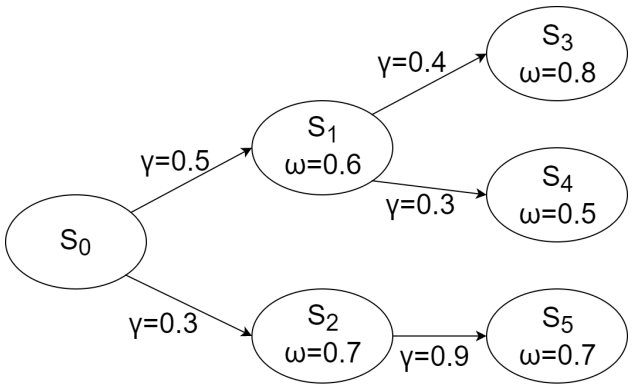


**Figure 4.4:** *A simple strategy graph to illustrate the possible plays from $S_0$.*

An interesting thing to note is that, even with a random selection of the play, it is possible for a planner to show the rationale behind the choices it makes by providing the values of $\psi$. In many domains the ability to show the rationale is paramount, as evidenced by [FLB+13], where an evidence based approach is used when explaining the result of a calculation. In our case the evidence provided is simply the calculated distribution, but it does elucidate the choices made by the planner. Another interesting aspect of calculating each distribution is that we can combine the two methods of planning (short-term and long-term). We accomplish this by considering an *experience value* $x \in \mathbb{R}$ such that $x \in [0, 1]$. If $x = 0$, then the team is completely inexperienced and uses short-term planning and if $x = 1$, the team is completely experienced and uses long-term planning. The intermediary values of $x$ should smoothly transition the approach from short-term to long-term. To accomplish this we do the following:

- Let $\{p_1, p_2, ..., p_n\}$ be the set of all plays available and, $\forall i \in [1, n]$:
  - Calculate the probability $P_s(p_i)$ using the short-term approach.
  - Calculate the probability $P_l(p_i)$ using the long-term approach.
  - Let $P(p_i) = (1 - x)P_s(p_i) + xP_l(p_i)$.
- Randomly choose a play using the calculated $P$.

### 4.1.3 The Realization Function

The *realization function* is responsible for ensuring that a play calculated by the planning function can be applied to the situation in the original domain and still accomplish the same behavior specified in the TSM. There are two parts of this adaptation that need to be considered, which we will call *corrections*:

1. The matching correction: This correction aims to fix the error introduced by the matching function. The amount of error introduced is measured by the $\mu$ metric defined in 4.1.1.

2. The planning representative correction: This correction aims to fix the error introduced when we pick a single representative state $\alpha_r$ to represent a whole class of situations when defining a ST in 3.1.2.

There are several approaches that can be employed to make the necessary corrections, but they are variations between adapting the situation to fit the play or adapting the play to fit the situation. In order to adapt the situation we must add some actions to the beginning of the play so we can reach a more compatible state. In order to adapt the play we must change the actions within it so that they are compatible with the current situation. Each possible approach might be more indicated depending on the domain and the kind of behavior we wish to simulate. We defined the following approaches to deal with the correction problem:

1. Double correction: Create an action to transition the original situation into one that fits within the equivalence class of the first SM in the play. Then create another action to transition that situation into the one picked as the representative in the first ST. Then simply follow the list of actions of each ST in the play as is. This approach first takes the system to the equivalence class of the beginning of the play, then it goes to the first representative and then if follows the calculated play as is. This simulates a very methodical behavior of seeking known situations to apply already calculated solutions.

2. Single correction: Create an action to transition the original situation into the one picked as the representative in the first ST. Then simply follow the list of actions of each ST in the play as is. This approach is similar to the double correction, but the behavior simulated is able to aggregate two corrections into one when positioning the system in a known situation.

3. Incremental correction: Calculate the relative distance between the original situation and the representative in the first ST. Then edit each action in the play to be applicable to the current situation while incrementally diminishing the relative distance between it and the next representative state. This approach adapts the whole plan to avoid drastic initial corrections, so the simulated behavior prefers to spread corrections over the whole play. The final state is still the representative state of the destination SM of the last ST.

4. Relative play execution: Calculate the relative distance between the original situation and the representative in the first ST. Then adapt each action in the play to be applicable to the current situation while keeping the relative distance between it and the next representative state. This approach adapts the whole plan to be applied to the current situation, leading to a situation that is as different from the representative of the destination as the initial situation was from the first representative. The simulated behavior considers the actions more important than the particular states achieved.

5. Matched incremental correction: This approach is identical to the incremental correction (as it incrementally leads to the representative of the final destination), but it first adds an action to transition the original situation into some state within the first equivalence class. This approach considers that the equivalence classes are important but any state within them can be worked the same way. The simulated behavior still wants to finish a play in a familiar state.

6. Matched relative play execution: This approach is similar to the relative play in the sense that it tries to keep the actions equal to the original play. However, it first corrects the initial situation with an action and it also ensures that the states found during the execution always belong to the equivalence class of the SMs in the play. This approach considers the equivalence classes to be important but any state within them to be equal.

Figure 4.5 illustrates the difference between each of the approaches listed above. It is worth noting that the only approach that does not necessarily leads to a state that belongs to the equivalence class of the last SM in the play is the relative play execution. While this approach does manage to keep the exact same action format of the original play, it ends up disregarding the equivalence classes, which might defeat the purpose of the defined strategy.

Every adaptation made by the realization function uses the domain restriction mentioned in section 3: there is always an action to control the value of a variable in $V$. In continuous domains

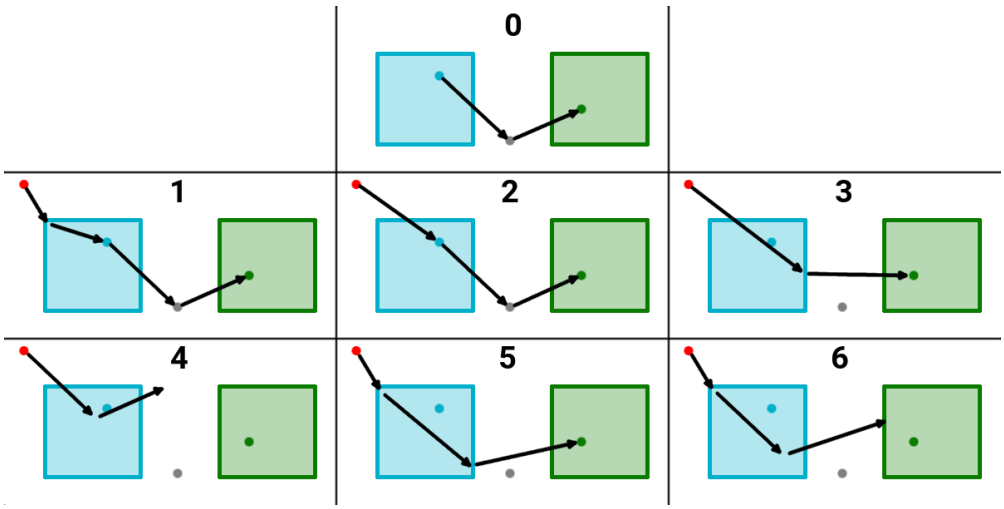


**Figure 4.5:** *The six different approaches to the realization of the play described in the frame 0: Frame 1 shows the double correction; Frame 2 shows single correction; Frame 3 shows incremental correction; Frame 4 shows relative play execution; Frame 5 shows matched incremental correction; Frame 6 shows matched relative play execution.*

this means that there are parametric actions to control the continuous variables, for instance: in a domain where there is a variable $v \in \mathbb{R}^3$ to control an object's position in space, there will be an action $a \in A(s)$, $\forall s \in S$ that changes that object's position to some arbitrary $x \in \mathbb{R}^3$. We will now formalize the process used to make each adaptation used by the approaches described above. As we have done in 4.1.1, let's define a SM by:

- A set of omitted variables $O = \{o_1, o_2, ..., o_k\}$.
- A set of grouping intervals $I = \{q_1, q_2, ..., q_l\}$, with components $q_i.a$ and $q_i.b$ being the start and end bounds respectively.
- Such that $k + l = n$.

There are 3 types of corrections that we use, and the following will describe how each correction can be accomplished:

1. Adapt $x \in S$ to a SM: for every variable $v \in V$ such that $v \in G$ with order $<_G$ and $x.v$ is the value of that variable in $x$:
   - If $v \in O$: nothing needs to be done.
   - If $v \in I$ and $q_v.a \leq_G x.v \leq_G q_v.b$: nothing needs to be done.
   - If $v \in I$ and $x.v \leq_G q_v.a$: create an action with outcome such that $x'.v = q_v.a$ when it is applied to $x$.
   - If $v \in I$ and $x.v \geq_G q_v.b$: create an action with outcome such that $x'.v = q_v.b$ when it is applied to $x$.
2. Adapt $x \in S$ to $y \in S$: for every variable $v \in V$ such that $v \in G$ with order $<_G$ and $x.v$ and $y.v$ are the values of that variable in $x$ and $y$ respectively: create an action with outcome such that $x'.v = y.v$ after it is applied to $x$.
3. Adapt $a \in A(y)$, $y \in S$ to $\{a_1, a_2, ..., a_n\} \in A(x)^n$, $x \in S$: This can be done in a relative manner (i.e., change both start and finish point of the action) or in an absolute manner (i.e., change only the start point of the action).
   - Relative: for every variable $v \in V$ such that $v \in G$ with order $<_G$ that is changed by $a$, create an action that creates this same change and apply it to $x$.

- Absolute: for every variable $v \in V$ such that $v \in G$ with order $<_G$ that is changed by $a$, create an action such that $x.v = y.v$ after it is applied to $x$.

Some of the approaches described also require an incremental approximation of a representative state from the play. In order to do this we need to be able to calculate intermediary states that smoothly transition one state to another. Fortunately, this calculation is analogous to the metric described in 4.1.1. One simply needs to break each state down to their variables and then interpolate each value, creating a new intermediary state by reassembling the variables. Discrete variables (e.g., $v \in G$ such that $v \in \mathbb{I}$) can be ignored from the interpolation or be interpolated and then approximated to a value in $G$. The decision of what to do should depend on the specific domain being simulated, but the worse case scenario is that an incremental approximation will behave like the relative play execution for these discrete variables.

## 4.2 Planning with the Basketball's TSM

The basketball domain was defined to control the collective planning during match simulations. The objective is to use it to extract a plan to be applied to any match situation. This planning should be executed anytime a player acquires the domain over the ball, in the case of basketball. In this chapter we will describe our method of planning that uses a basketball's TSM to calculate a specific plan to any match situation quickly and faithful to the input strategy. The basketball simulation domain we defined is in fact compatible with the definition in 3.1:

- The state space $S$, while infinite, can be described by a finite set of variables $V$:
  - A court position for each player (in $\mathbb{R}^2$)
  - A rotation for each player (in $\mathbb{R}$)
  - A court speed for each player (in $\mathbb{R}^2$)
  - A ball holder index (in $\mathbb{N}$)
  - A ball position (in $\mathbb{R}^3$)
  - A ball speed (in $\mathbb{R}^3$)
- Each state has a set of actions that can be applied to any of the players:
  - A Move action, that can change the position and speed of the player.
  - A Rotate action, that can change the rotation of the player.
- And the player with the ball can manipulate the ball (by actions Pass and Shoot).

Our planning method using the TSM allows us to calculate a plan in a simpler environment (the strategy graph of the TSM) and apply the calculated plans to the complex domain of the match simulation. We deal with the unforeseeable nature of the domains by assuming every action will execute as planned and re-planning whenever this assumption breaks. Such approach has been described in planning literature as a way to deal with uncertainty ([CYS94]). This happens whenever a ST fails to transition to the destination SM, when a pass is intercepted or the ball is stolen, for instance. The small size of the strategy graph allowed us to assume a constant re-planning need without worrying about performance issues.

There is potentially a big difference between what is happening in the match and what can be represented by the TSM, so we need to be able to translate between these two domains freely to be able to plan for any situation. We accomplished this by creating the matching and realization functions in our basketball domain.

In this section we will explain the basketball's matching function in detail in 4.2.1, then we will cover the planning process in the simplified domain of the TSM in 4.2.2 and finally we will end by explaining the realization function in 4.2.3. The presentation order reflects the order in which we

execute each function when calculating a plan. Figure 4.6 illustrates this process in the basketball simulator. Note that the matching process now provides the player allocation (i.e., which player will follow which player role in the TSM).

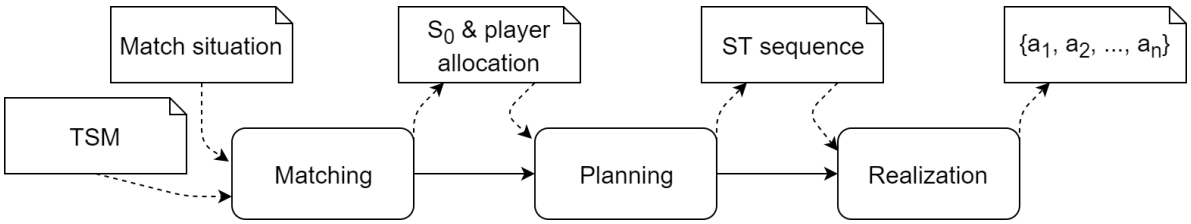


**Figure 4.6:** *The basketball's planning process, comprised of matching, TSM planning and realization.*

### 4.2.1 The Matching Function

This function is required because we have no guarantees that every possible match situation belongs to the equivalence class of exactly one SM in the TSM. It is possible that some situations belong to the equivalence classes of more than one SM and it is a near certainty that many situations will belong to none. It would be impossible to require that the basketball specialist providing the TSM constructs strategies with such properties, but fortunately we do not need them. The matching function translates any match situation into a SM in the TSM by defining a metric to compare match situations to SMs. This metric guarantees that, for any given match situation, a SM can be found in the TSM, even if some representation error is introduced.

First, we create a metric called *p-dist* that measures the distance between a position and a player role. The p-dist is used to measure how far a player in the match is from belonging to some player role in some SM. The p-dist is equal to the smallest distance between the position and any point in the role's playing area. If the position is outside of the playing area, this is the distance between point and ellipsis, as indicated in figure 4.7. If the position is inside the playing area the p-dist will be zero and we consider that the match player belongs to the player role.

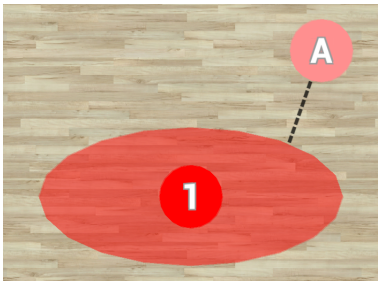


**Figure 4.7:** *The distance between match player A and player role 1, called p-dist, is indicated by the dotted line.*

We then create a metric called m-dist to measure the distance between a match situation and a SM. The m-dist is a composition of the p-dist for every player. We define two types of m-dist that change how this composition is done:

1. Temporal: in the temporal m-dist we want to select the SM where we can make every player belong to their role in the least amount of time. To accomplish this the m-dist is simply the largest p-dist, as that is the player who will take the longest to belong to his role.

2. Spacial: in the spacial m-dist we want to select the SM where we can make every player belong to their role with the least amount of movement. To accomplish this the m-dist is the sum of every p-dist.

During a simulation the temporal m-dist can be used when we want to quickly adapt to a strategy and the spacial m-dist will minimize player fatigue during adaptation. Figure 4.8 illustrates

the difference between the two types of m-dist. The result on the right side of the figure will take longer to adapt to the SM found due to the greater distance player C needs to traverse, but the total distance traversed by all players is smaller.

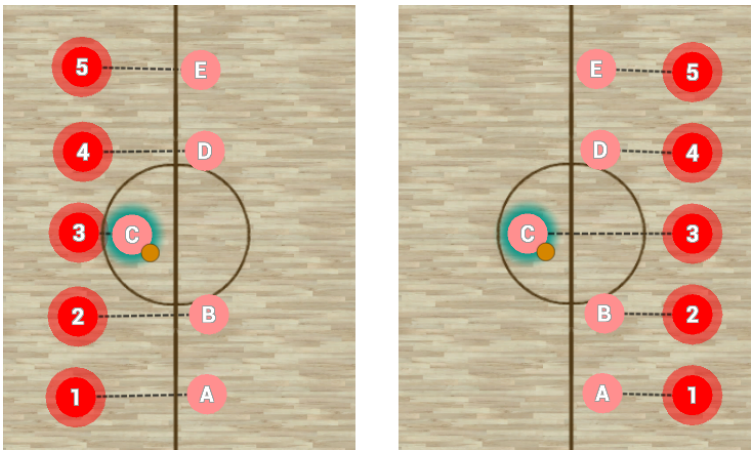


**Figure 4.8:** *Two different results of the matching functions applied to the same situation. The result on the left uses the temporal m-dist and the one on the right uses the spacial m-dist.*

One might note that we omitted the weight of the variables in our matching, this is done because we assign the same weight to each offense player and no weight no the defense players. Therefore, we can consider only the offense players and ignore the weight in the calculation. In any case, the m-dist greatly facilitates the implementation of the matching function. The function simply needs to search for the SM that minimizes m-dist among all the available in the TSM. We still need to consider how we pair the players to the roles when we calculate the metric (and when we apply the plan later). We have 3 ways of doing this, each with their own set of advantages and drawbacks.

### Player Allocation Methods

In a match with 5 offensive players, there are 120 possible player allocations ($5! = 120$) in a SM. We use 3 different approaches to find the allocation we want to use:

1. Fixed: An allocation is decided in the beginning of the match and we maintain it throughout the simulation. This method is useful if we consider that each player is very different from the next and the strategy defines specific roles with specific players in mind. This allows a strategy to be designed with specific player abilities in mind, such as height or shoot accuracy.

2. Greedy: The greedy allocation utilizes a greedy algorithm to generate the allocation. The algorithm calculates the p-dist for every player and role pair and allocates the pair with the smallest p-dist. This logic is repeated until every player is allocated to some role. In terms of behavior simulation, this considers that each player will assume the closest role that is not yet taken by a closer player.

3. Minimal: The allocation that minimizes the value that will be calculated by the matching function. Except for some optimization in the calculation, this algorithm essentially tests every possible allocation and returns the one that minimizes the m-dist. The type of m-dist (temporal or spacial) needs to be considered when testing allocations. While this method yields the most efficient allocation, it is not a very natural human behavior. It would be more appropriate for the simulation of robots, for instance.

It is not trivial to notice that the allocation found by the greedy method can be different than the minimal allocation, but figure 4.9 contains an example where that happens. The figure shows what allocation each method would calculate for the given situation and SM.

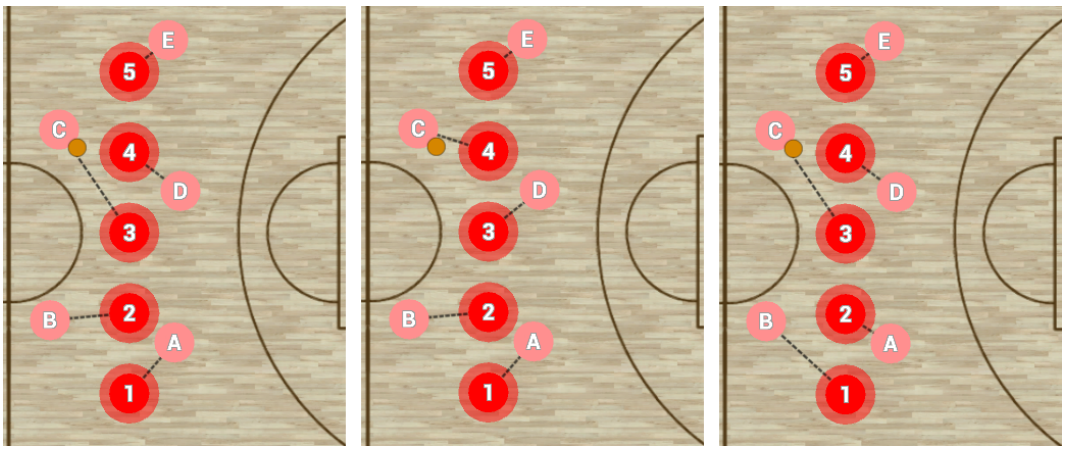


**Figure 4.9:** *An example of the three allocation methods. From left to right we see the fixed, minimal and greedy allocations.*

### 4.2.2 The Planning Function

As mentioned in 4.1.2, the planning function does not seek to find the best possible play in the TSM every time. We wish to implement a function that picks different plays using a reasonable method given the information in the TSM. This means that a team during a simulation will favor plays that are safer and yield greater rewards, but still execute riskier or lower reward plays sometimes. The goal is to ensure that the behavior of the team is more realistic and not completely predictable.

While still prioritizing the best plays, this method allows for a greater use of the strategy graph (i.e., more varied plays will be executed). This feature allows the user to see his whole strategy being simulated more easily and it facilitates automatic learning by simulation. During a simulation one might wish to enhance the risk and reward estimations, and it cannot be done for the whole graph if a play is never executed.

We use the method described in 4.1.2 to score the possible plays and select one to be executed at any given moment. The technique to smoothly transition between planning methods (short-term and long-term) makes sense in a basketball domain. As a team and its players gain basketball experience we want to transition from short-term to long-term.

Regardless of the method used to pick plays, the specified value of risks and rewards play an immense part in the behavior that is simulated. We can receive those values from the designer and use them as is, even if the values are incorrect. Doing this will simulate the specified TSM, including the designer's misconceptions. This might be the desired result, but we might also offer to use the simulation to enhance the TSM by learning more accurate values for the risks and rewards.

In any case, the final result of the planning is a path in the strategy graph. This sequence of STs fit together smoothly and they contain movement, passes and synchronization for every player, as described in 3.2.2. Figure 4.10 shows the information contained in a play (zoomed to show a single player).

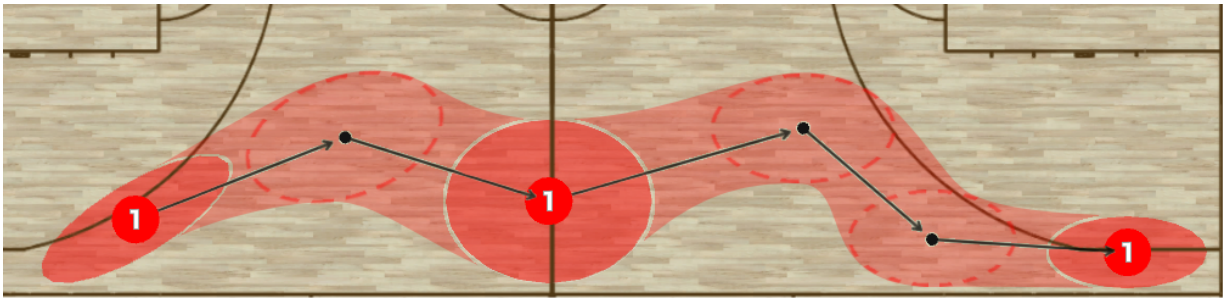


**Figure 4.10:** *The information contained in a play with 2 transitions, zoomed to show a single player.*

### 4.2.3 The Realization Function

The realization function needs to adapt the play found by the planning function to be applicable to the current match situation. As described in 4.1.3, there are many approaches to do this. We consider the equivalence classes to carry important strategic value in basketball, so we consider only 3 of the 6 approaches in this domain: the double correction, the matched incremental correction and the matched play execution.

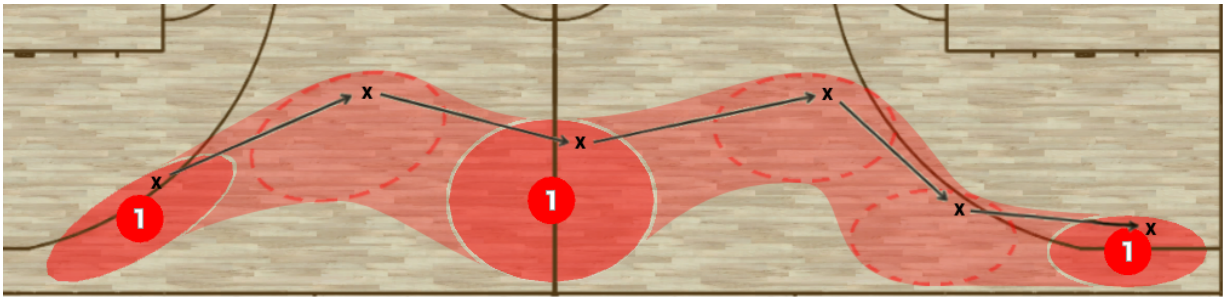


**Figure 4.11:** *Relative adaptation of a play.*

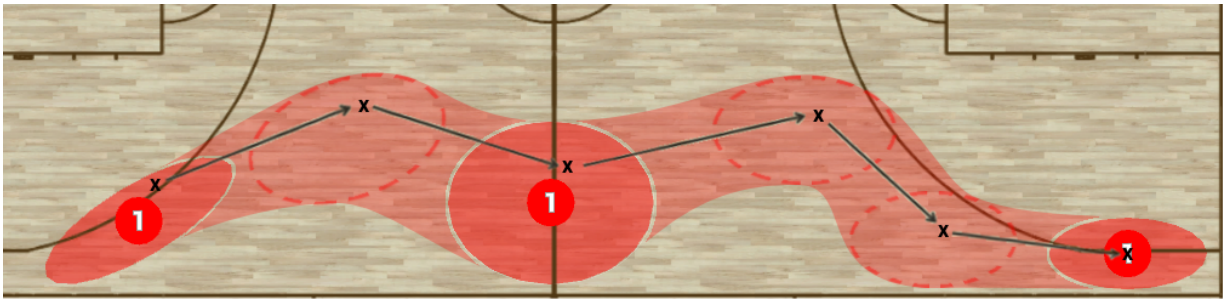


**Figure 4.12:** *Incremental adaptation of a play.*

The necessary adaptations are trivial in this domain. Every action that needs to be adapted is a player movement action and all relative positioning and interpolations are done in $\mathbb{R}^2$ using simple geometry. We need to correct for every player, so some synchronization is required for each player to begin the play at the same time. Quite simply, we start by generating actions to bring each player inside their first play area of the plan. We select speeds or add wait times to ensure every player is ready to begin at the same time. After each player is in the correct play area, we adapt the actions in one of three ways:

1. Centralizer: We bring each player to the center of the play area and execute the play exactly as described in the STs. This is the double correction approach from 4.1.3. While this is the easiest approach to implement and understand, it removes from the diversity of the simulation and the behavior becomes somewhat robotic. It is useful to recognize exactly the plays being executed but it carries little value other than that.

2. Relative: Once inside the play area, the player keeps his relative position from the center while executing the whole play. The relative position is maintained by keeping two variables constant during play: 1 - the angle relative to the rotation of the ellipsis; 2- the ratio between the distances from player to the center and from edge to center of the ellipsis. This means that the absolute distance between the player and the center of the play areas will change depending on the size of each area. Figure 4.11 shows an example of this.

3. Incremental adaptation. During the play a player will incrementally move towards the center of the play areas, finally reaching it by the end of the play. This behavior emerges when the players try to gradually move towards the strategy in the TSM. Figure 4.12 shows an example of this.

The result of the realization function, regardless of approach, is a sequence of actions for each player that fits exactly with the current match situation and follows the intent of the TSM. This play contains general movement and synchronization as well as ball handling actions. Chapter 5 will show how this information can be incorporated into a system that controls more layers of the simulation, such as individual adaptation and mechanical simulation.

# Chapter 5

# Layered Control System Architecture

We developed a layered simulation control architecture capable of implementing several levels of control over the agents in a modular and hierarchical manner. The simulation of a team of agents working with the same objective begins at a high level collective planning layer and ends with the correct placement of body parts (e.g., hands and feet) of each agent's 3D model. We work with 3 layers of control: collective, individual and mechanical, as well as a physical simulations layer between the individual control and the mechanical. The collective layer is responsible for the implementation of the high level collective planning, doing so uses the concepts presented in chapters 3 and 4. This layer provides a high level plan to be executed by each of the teams in the simulation. This serves as input to the individual control layer, which adapts each agent's actions to their individual perception, behavior and constraints. Details not considered in the collective plan, such as obstacle avoidance, get implemented in this layer. This layer yields a more detailed list of actions for each agent to attempt to execute. The physical simulation layer actually executes the planned actions, simulating results and changing the state of the world. Finally, the mechanical layer is responsible for placing the agents' limbs in a manner consistent with the physical simulation results.

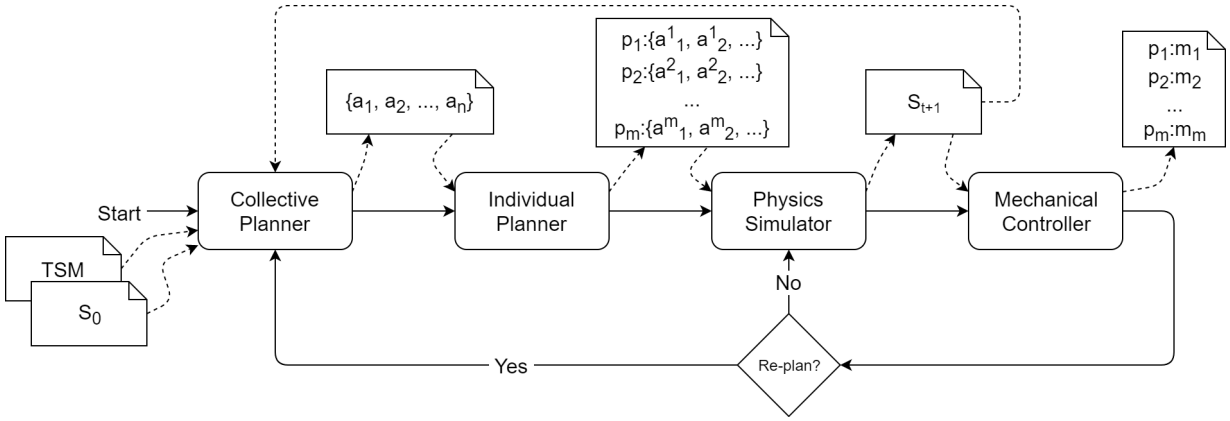


**Figure 5.1:** *Layered architecture for simulation of a team of agents working in cooperation.*

Figure 5.1 gives an overview of the layered architecture. The original input is a TSM and the initial simulation state $S_0$, then the simulator enters a loop of simulation that generates the next simulation state $S_{t+1}$ in every iteration of the loop. The figure represents the simulation of a team of $m$ agents: $\{p_1, p_2, ..., p_m\}$. The collective planner starts the process receiving the initial state $S_0$ and the TSM, then it generates the team actions $\{a_1, a_2, ..., a_n\}$. Using the team actions the individual planner calculates a different list of individual actions for each of the agents: $p_i : \{a_1^i, a_2^i, ...\}, \forall i \in [0, m]$. The physics simulator executes each individual action and calculates the next state of the simulation $S_1$. The mechanical controller then uses $S_1$ to generate a model for each agent:$p_i : m_i, \forall i \in [0, m]$. These models are the graphical representation of the situation

of each agent. Finally, the simulator reviews the state of each action on the plan and evaluates whether or not a re-plan is needed. If a re-plan is needed $S_1$ is sent to the collective planner and the loop starts from there, otherwise the physics simulator continues simulating the next actions in each individual action list.

This architecture can be adapted to control any number of teams of agents in the simulation, each team using one or more TSMs. In sport simulations it is usefull to have different TSM for the offensive and defensive behaviors, for instance. The defensive and offensive TSMs are defined for the basketball domain in 3.2 and 3.3. In this case, each team has a collective planner and each planner decides which TSM to use depending on the current simulation state $S_t$. Figure 5.2 shows a two team match simulation architecture.

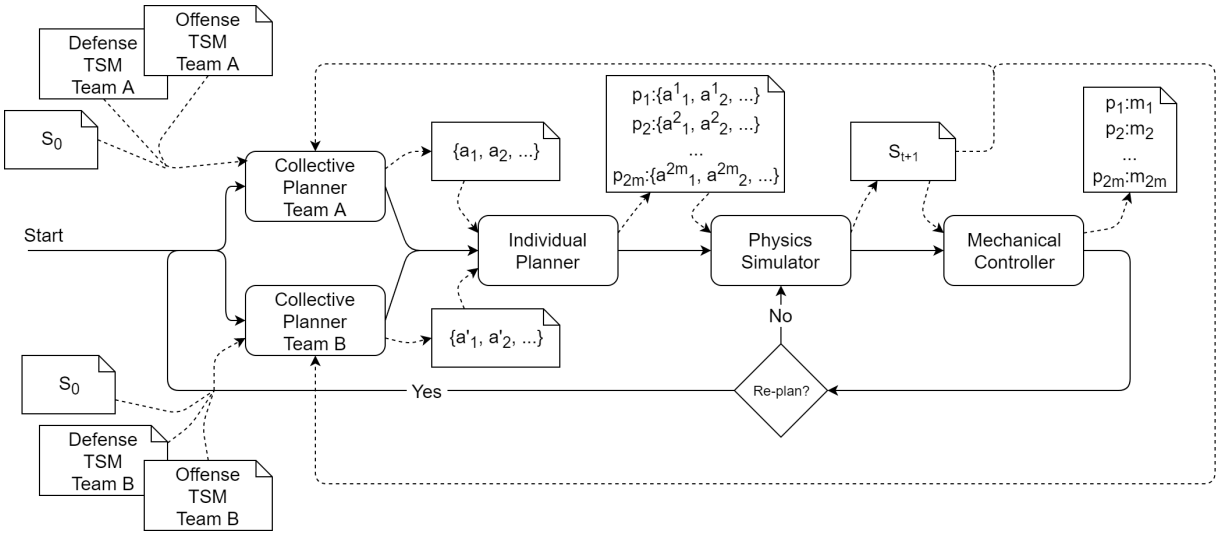


**Figure 5.2:** *Layered architecture for simulation of the competition of two teams of agents working in cooperation.*

It is worth mentioning that the states $S_t$ of the simulation can also carry meta-data about the simulation that could impact the planners (both collective and individual). The match score, for instance, could change the current sub-graph of the strategy being used. Another important meta-data is the judgment of the actions according to the rules of the domain, some of which could trigger different behaviors. An example of this would be a foul in a basketball simulation that would trigger a free-throw, where each player needs to be positioned in specific places and wait for the shot. In [Ros97] an approach to action selection is described as a vote by different modules in the agent. We consider that a rule following module is added to each agent and their vote carries more weight. The initial state $S_0$ also carries meta-data relative to the beginning of the simulation, which in basketball would trigger the players to position themselves for the tip off that starts a match.

## 5.1 Collective Planner

The collective planner can work in an asymmetric manner, employing a different set of TSMs for each team of agents being simulated. In order to use more multiple TSMs, one must define clear rules to choose among the available TSMs at any point during the simulation. In the basketball domain, for instance, the defense TSM is used when the opponent has the domain of the ball and the offense TSM is used in every other situation.

It is important to note that the approach of using more than one TSM per team is not the same as defining a state machine of sub-graphs, as described in 3.1.3 and 3.2.3. The sub-graph approach will allow the user to define different portions of the strategy to be used at different times, but the matching, planning and realization functions will remain the same. Defining different TSMs allows

us to change the way we implement the matching function, as well as the planning and realization functions. Two TSMs defined on the same domain can still behave very differently, as we can see by the basketball examples in 3.2 and 3.3. Figure 5.3 illustrates the collective planning process.

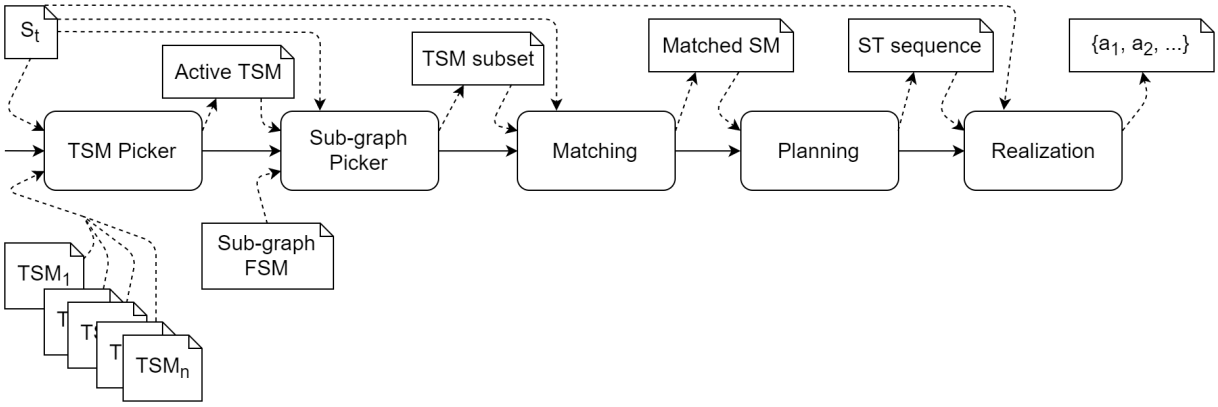


**Figure 5.3:** *The collective planner process detailed to show multiple TSM and sub-graph handling.*

The collective planner starts by using $S_t$ to decide among the available TSMs in the TSM picker process. Then the sub-graph picker process uses $S_t$, the active TSM and a sub-graph FSM to decide which subset of the TSM should be used. Finally the matching, planning and realization process worked as described in 2.1.

### 5.1.1 The basketball's collective planner

The collective planner for a basketball team uses two TSMs, the offensive and the defensive. Of course, there are two collective planners, one for each team. Each process does the following in the basketball's collective planner:

1. TSM Picker: This process simply checks in $S_t$ if the opposing team has domain of the ball. If that is the case it picks the defensive TSM, otherwise it chooses the offensive TSM.

2. Sub-graph Picker: This process updates the FSM that chooses the sub-graph by using $S_t$ to verify if any transitions need to be executed. It then extracts the TSM subset from the active state in the FSM.

3. Matching:

   - Offensive TSM: The matching finds the ideal player allocation and the SM in the active subset that minimizes the metric being used, as described in 4.2.1.
   - Defensive TSM: The matching finds an allocation of defense players to offense players. Then it finds one equivalence class for each defender, looking at the defender's allocated target and the ball holder. The possible equivalence classes are described in 3.3.1.

4. Planning:

   - Offensive TSM: The planner finds a path in the strategy graph with a sequence of ST that contain the collective actions that need to be executed. This process is detailed in 4.2.2.
   - Defensive TSM: For each defender, the planning process will find the ideal direction, distance and gaze in the defensive TSM. Then it will create collective actions that take all players to their ideal places. A detailed explanation of the information contained in the TSM is in 3.3.1.

5. Realization:

- Offensive TSM: The realization creates the specific actions for each player according to the play selected by the planning function, as described in 4.2.3.
- Defensive TSM: The realization simply creates the specific actions for each defender by assigning the correct parameters in the parametric actions described by the TSM. It will calculate an exact point by checking the direction and distance parameters and create a move action for that point, for instance.

## 5.2 Individual Planner

The individual planner creates individual actions that can be easily executed by the physics simulator and will: 1 - respect the intent of the collective plan; 2 - respect the limits of the agent and situation. While the specific actions and adjustments done in the individual planner are domain specific, the general architecture is always the same. An individual planner will have an array of processes for each agent in the simulation, each array doing the necessary work to create the individual action sequence for their agent. Figure 5.4 illustrates this process.

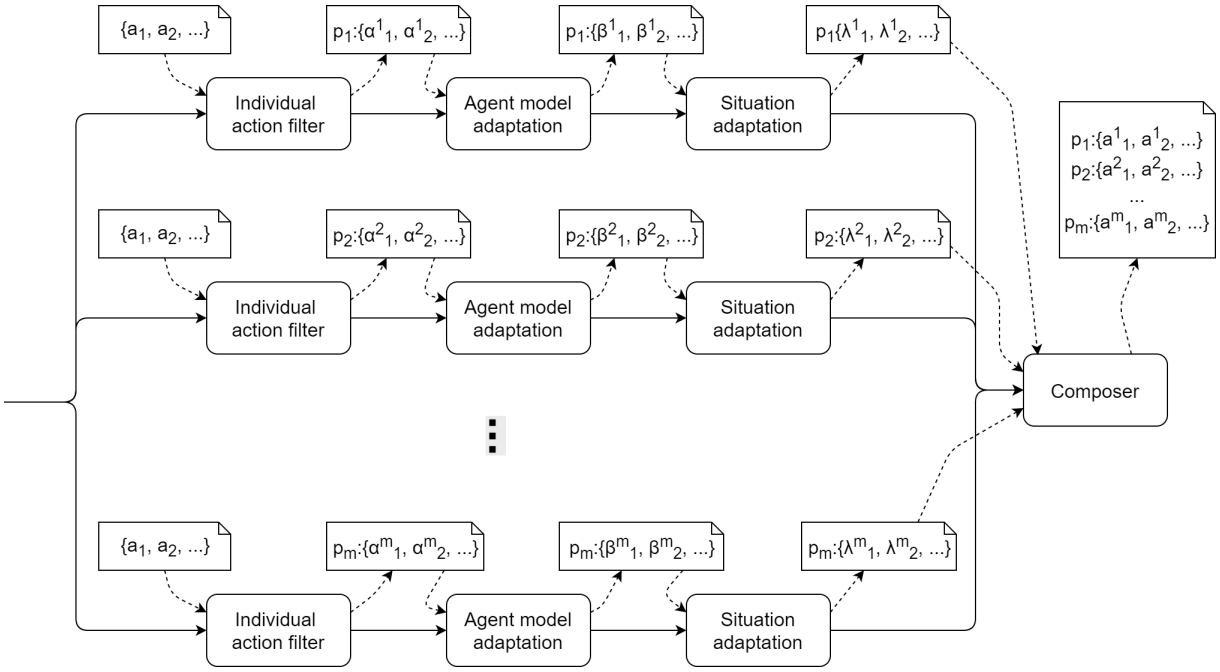


**Figure 5.4:** *The individual planner process for a group of m agents.*

The array of process for each of the agents consist of the same 3 steps:

- Individual action filter: This process filters out the agent's actions from the collective plan. A collective action that requires more than one agent will be broken into individual parts. For instance, a pass ball action might be broken into send ball and receive ball actions. The result of this process is a list of individual actions for the agent.
- Agent model adaptation: This process adapts each action on the list to be compatible with the model of the agent executing them. A movement action might become a run or a ride horse action depending on the type of agent in a war simulator, for instance. The resulting list of actions are compatible with the agent that will try to execute them.
- Situation adaptation: This process adapts the actions to the situation in which they will be applied. This could mean adding obstacle avoidance or changing the speed to match different terrains in movement actions. The result is a situation aware list of actions for the specific player to execute.

Followed by a composer that assembles every action into a single structure and ensures proper reactions are added to players where necessary. A reaction is an action that responds to some other action executed by another agent. If an agent executes a strike action, the system might require the component to add a get hit action to the list of the target. This get hit action would deal with the results of being the target of a strike action (e.g., check for damage, delay next action, etc).

The individual control layer allows for a more sophisticated simulation of complex behavior without requiring too much from the TSM. It would not be reasonable to require that obstacle avoidance be added to the TSM in most cases, but implementing it in the individual layer can enhance the simulation without burdening the strategy designer.

The actions resulting from the individual planner only need to consider the domain attributes necessary to drive the physical simulation. Attributes relevant to other aspects of the simulation that do not concern the physical simulation of the domain are ignored in this layer, things such as 3D animation, sound, visual feedback for the user, etc.

Finally, the actions created by the individual planner need to be ready for execution by the physical simulator. They need to implement a simple interface with a *tick* function, that simulates the execution of the actions for a given amount of time and returns one of 3 values:

- Executing: the action executed for the given amount of time but has not completed yet.
- Success: the action finished executing successfully. This result also returns the amount of time remaining (i.e., not used to complete the execution).
- Fail: the action could not complete and failed its execution. This result also returns the amount of time that was not used.

In section 5.3 we will describe how this interface is used to drive the physical simulation.

### 5.2.1 The basketball's individual planner

The individual actions in our basketball simulation are all related to movement or ball handling. In the individual layer the players are considered to by cylinders with a direction, meaning there is a side we consider to be the face. These cylinders have a position, a rotation and a speed, and they can manipulate the ball by carrying, grabbing or throwing it. The result of this simple model are actions that manipulate the player's facing and velocity and some high level actions with the ball. This is enough to implement sophisticated strategic behavior and all we really need to simulate in the physical layer. The graphical realism we would like to see in a sport simulator comes from the work of the mechanical controller, described in 5.4.

The actions of our basketball simulation are as follows:

- Move: A simple movement action. At every tick the action will attempt to move towards some designated target position with some designated speed. In the basketball domain this action never fails, so it will return *executing* until it reaches the target, then it returns *success*.
- Wait: A simple wait action. At every tick the action will decrease the wait timer and do nothing else. This action cannot fail in the basketball domain, so it simply returns *executing* until the timer runs out, then it returns *success*.
- Pass: A pass is executed by the ball holder with the objective of giving the ball to another player. If this action is executed when the player has the ball, it will simply send the ball off and return *success*. If the execution comes at a time when the player is not in possession of the ball, the pass action will return *executing* for a small amount of time while it waits to see if the ball reaches the player. If the time runs out and the player still does not have the ball, the action returns *fail*. This small tolerance allows the team to continue executing a play even when some small delays occurs.

- Receive: The player executing the receive action tries to take possession of the ball. This action might fail if the player is not near the ball or by a simple fumble when trying to reach for it. Like the pass action, a small time tolerance can be implemented.

- Shoot: The shoot action is executed by the ball holder with the objective of scoring a point. If the shoot action is executed when the player has possession of the ball, the ball will be shot and the action returns *success*. Otherwise a time tolerance is implemented just like the pass action. Whether or not the shot goes in the basket, if the ball is successfully thrown, the action will return *success*.

- Steal: The steal action is an attempt to take possession of the ball from an adversary. This action fails if there is no adversary with the ball nearby or if the steal attempt is made and fails. A success means the steal attempt was made and the ball was taken.

The adjustments made by the individual planner in this domain are mainly in the movement actions. The individual planner can make adjustments to the designated speed specified in the TSM to seek more realism in the simulation. The TSM simply specifies a speed for each movement action, no acceleration of deceleration. The individual planner can create variations in speed that allows for smooth movement while trying to maintain the same average speed specified in the TSM. Some wait times might also be adjusted if the player starts lagging behind in the play execution due to the movement speed adjustments.

Variations of these actions can be implemented to give more sophistication to the simulation. The shoot action can become a dunk when the player is under the basket. The steal action can became an intercept if the ball is flying through the air during a pass. The pass and receive actions can become handover and take actions when the players are near each other. Each new action or variation brings more depth to the simulation, but the essential effects are those of the main actions: change the player positioning, switch ball holder within the team, score attempt and change ball holder between teams.

## 5.3 Physics Simulation

The physical simulation layer runs through every agent in the simulation requesting the execution of the first action in their list. This layer is also responsible for determining the result of actions that require some form of simulation (e.g., a dice roll). The work done for each agent is similar to the execution of a behavior tree, described in 2.2.2. The actions that return *executing* are kept on the agents' lists, actions that return *success* are removed and actions that return *fail* are analyzed and might trigger a re-plan in the next simulation cycle.

While this layer is called physics simulation, it receives this name because physical attributes of the environment are update by this simulation, not because it necessarily simulates the actual physics of the actions. For instance, a dice roll might be decided by reading some random variable, not by actually simulating the dynamics of a small painted wooden cube being thrown on the ground.

This layer is also responsible for enforcing the rules of the domain and external events. For instance, if there is some time limit for the simulation (e.g., a round timer in a box match), the physics simulation will generate the necessary meta-data in the simulated state that indicates the time and whether or not it has run out.

Figure 5.5 illustrates the process implemented by the physics simulator. The process is comprised of 3 steps:

1. Tick actions: For each agent in the simulation, the simulator ticks the first action in their queue. Actions that require simulation have their results calculated (possibly using the executing agent's model of abilities). The state $S_t$ is modified to aggregate the results of the actions, becoming $S'_t$.

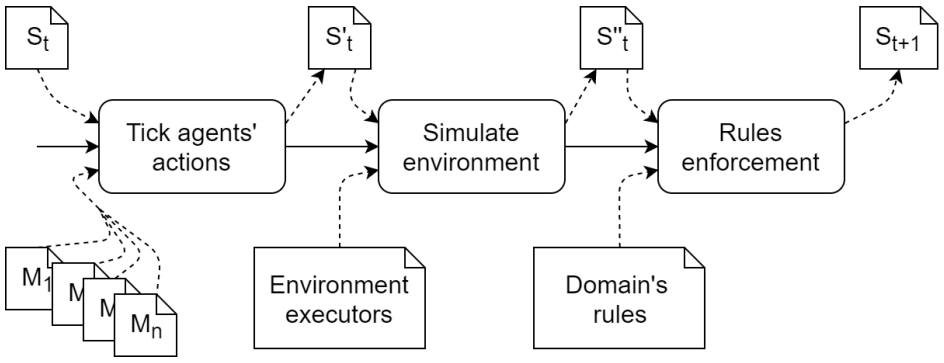


**Figure 5.5:** *The physics simulator a group of n agents with models* $\{M_1, ...M_n\}$.

2. Simulate environment: This process simulates the effects of every non-intelligent entity that might execute some form of action in the environment (e.g., a volcano erupting, an automatic door opening). The list of environment executors is used for this purpose and the state is further modified, becoming $S_t''$.

3. Rules enforcement: The rules enforcement ensure the domain's rules are followed and updates the state to reflect the judgment of said rules. It also adds meta-data to the state to reflect the need of re-planning based of the results of each action executed in the two previous processes. The final result is the state $S_{t+1}$, which contains the updated state of the world, the re-plan flag and any rule judgments that might impact future planning (e.g., a foul in a sport simulator).

### 5.3.1 The basketball's physics simulation

The basketball's physics simulator ticks the actions for every player in both teams and updates the state to reflect their impact, as well as any external effects and rules. In our simulator the only environment executor is the ball, and only when it is not being dominated by a player. The basketball's domain rules are basic physical world rules (e.g., two players cannot occupy the same space at the same time, gravity) and basketball rules (e.g., ball out of court means side-throw).

The uncertain actions in basketball, such as score attempts, are simulated using simple player models. For each player we have a score that governs how well he executes an action and it is used when randomizing the result of executing said action. For instance, a score that estimates the likelihood of a player succeeding in a 3-pts shoot is used when that player attempts such a shot. One can also use the current match state to calculate modifiers for each score. For instance, a marked player is less likely to score a 3-pts basket. While the proposed system supports any number of player attributes and modifiers, the specification of these has been left as future work.

After ticking the players actions, the physics simulator will update the ball position (in case it is not with some player) and go onto rules enforcement. The rules enforcement ensures no players are occupying the same position and checks if any basketball rules need to be enforced through state meta-data. An example of a rule enforced by meta-data is the point score: if a team scores the simulator will add a flag that will cause a player from the other team to take the ball outside before restarting the match.

## 5.4 Mechanical Control

The mechanical control is responsible for the graphical fidelity of the simulation. We call it a mechanical control because this usually means the correct placement of the agents' limbs, but this layer also creates any other audio/visual feedback necessary for realism. The mechanical control layer executes after the physical simulation because the effects created do not impact the physical state of the world, they are purely cosmetic.

The mechanical layer allows for a greater realism of the simulation, even without directly impacting the state of the environment. This is true because the physical simulation cannot reach the level of detail required to make the cosmetic effects of this layer unnecessary. Take a humanoid agent walking as an example of this. The physical simulation will move the agent's body in the correct speed and the mechanical simulation will then play an animation of the agent's body moving the legs in the correct speed. Now, if the physical simulator actually calculates the position and rotation of each member of the agent's body, instead of considering it a single entity, the animation done through the mechanical simulation would be unnecessary. Currently such depth of control in the physics simulation is unpractical and mostly unnecessary, so a final mechanical layer is added.

The continuous calculation of the movement of each member in the body of an agent is an immense challenge that very frequently yields poor results. The industry usually works with a predetermined set of animations and adapts them through blending and IKs to fit the current state of the simulation. We have covered the most common approaches to animation blending and IKs in 2.3.

On top of the animation control of the agents' bodies, the mechanical layer also creates the necessary audio/visual effects to more adequately reflect the state of the simulation. This might mean playing an audio when some loud action occurs or changing the lighting situation when an action calls for it (e.g., a lamp is lit, the sun rises).

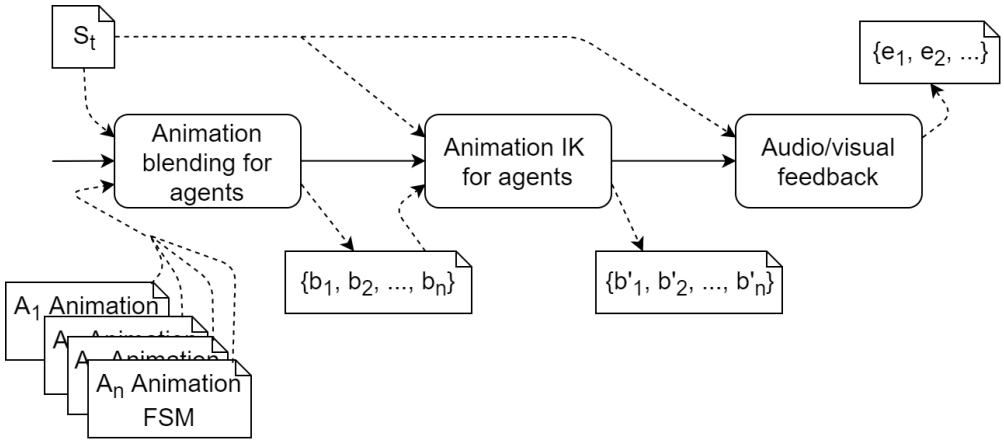


**Figure 5.6:** *The mechanical control layer of a simulation with n agents.*

Figure 5.6 illustrates the mechanical control layer. It is comprised of 3 steps:

1. Animation blending: This process updates the animation machine for each of the agents using the information contained in $S_t$. The resulting animation blend reflects the general movement being executed by each of the agents in the simulation.

2. Animation IK: This process tweaks each blend of animation to place the agents' limbs in a manner consistent with the state of the simulation. The placement of feet around terrain imperfections or placing hands exactly on the object being held are examples of the use of animation IK.

3. Audio/visual feedback: Events in the environment that require some form of audio or visual feedbacks are calculated in this process, which generates a list of effects $\{e_1, e_2, ...\}$. An event in the environment might be caused by the actions of an agent or some environmental executor. As an example, a lit torch might generate 3 different effects: a visual particle effect of smoke and fire, a visual light effect and a continuous burning sound.

### 5.4.1 The basketball's mechanical control

In our basketball simulator the main job of the mechanical control layer is to control the animation of each of the players. Some of the events do need audio and visual feedback, but those

are secondary in our simulation. The effects accomplished by each of the processes in figure 5.6 in the basketball simulator are:

- Animation blending: This ensures that each player is using an animation that is compatible with their movement speed and direction. The process can blend between 3 speeds (standing, walking and running) and 4 directions (backwards, left, right and forward) to accomplish the desired result. There are also 2 layers with masks, one for the feet and another for the arms and torso. This is used because the player with the ball moves his arms differently to conduct the ball.

- Animation IK: This is used only for the player with the ball. We place an IK chain on the dominant arm of the player to ensure that the hand exactly meets the ball at the appropriate times of the animations.

- Audio/visual feedback: There are two important visual effects in the simulation, the current match time and the score. These are updated as clues in the graphical user interface. Any other effect deemed necessary (e.g., sound of the ball hitting the ground) can be added to this process as required.

It is worth mentioning that the exact setup of the animation FSM is a more artistic than mathematical process, but there are ways to assist the animation designer. A common way to do so is to analyze the speed of movement contained in a clip of a walk cycle or a run cycle. This is done by calculating the space traveled between steps and the playback speed of the clip. This sort of tool can facilitate the setup of the animations FSM, but it is usually a job ultimately left in the hands of an animation designer.

# Chapter 6

# Basketball Simulator Software

We have created a software to test and validate the concepts developed during our work. This simulator software implements the architecture described in chapter 5 for the basketball domain. We also implemented a strategy design tool for both offense and defense in basketball. In the same system the user can design a TSM, test the results of the planning on that strategy and see the effects in a match simulation.

We used a game developing engine to create the software to accelerate production time and allow the results to be deployed to all major platforms. We chose to develop in the Unity3D engine due to the large amount of existing assets and supportive community, as well as the affordable price range (that starts with a free version). The software was developed using the *C#* programming language in a manner that is both expandable and easy to understand. The remainder of this chapter will be dedicated to the description of the implementation of every aspect of the simulator software. Section 6.1 begins by explaining the TSM designer for the offense. Next we talk about the use of the defensive TSM in section 6.2. Then, each layer of the control system architecture will be described: the collective planner (6.3), the individual planner and physics simulator (6.4) and the mechanical control layer (6.5). Finally, section 6.6 will describe the results achieved by the simulator: the input, the configurations and the output.

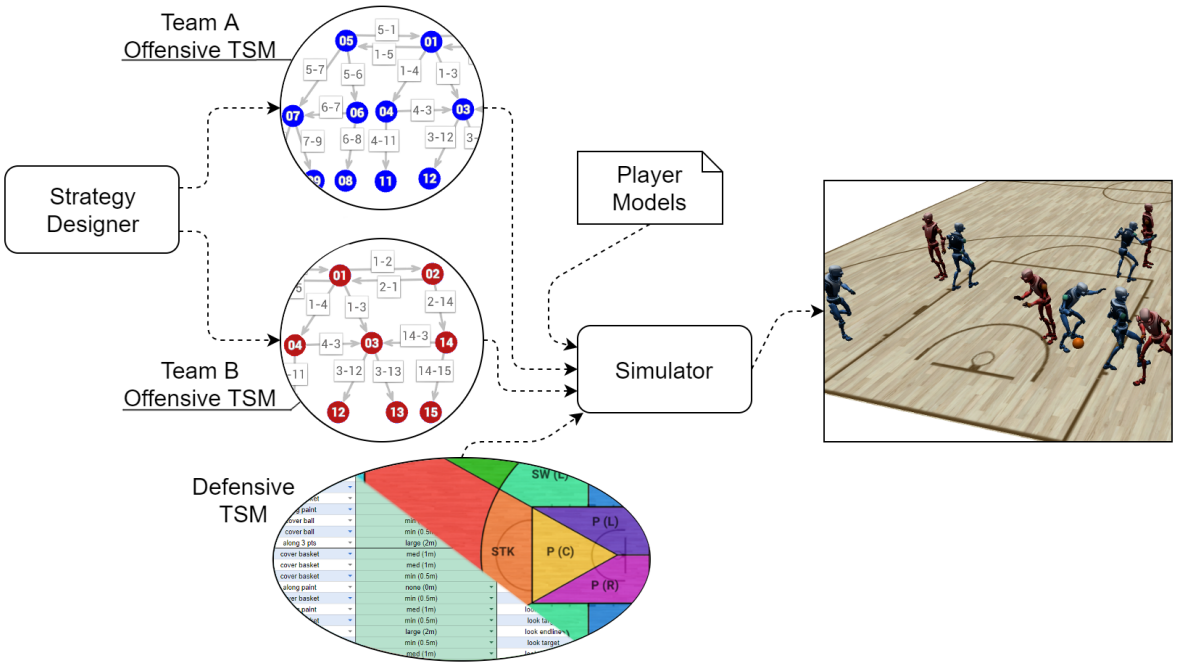


**Figure 6.1:** *The input and output of the basketball match simulator.*

Figure 6.1 shows an overview of the input and output of our simulator. As input the simulator takes two offensive TSMs that are created by our design tool, a defensive TSM currently shared

by both teams and a collection of player models. The output is a graphical representation of the simulation.

## 6.1 Strategy Specification Tool

The strategy specification tool is a helper tool that allows the basketball specialist to quickly create TSMs. Since the user base of the tool are basketball specialists (e.g., coaches, sport enthusiasts), we could not required a deep knowledge of computer systems. We tried to create a tool that was easy to learn while still providing the necessary shortcuts to accelerate the design.

The tool saves each TSM in a simple XML file that can be shared through familiar methods (e.g., e-mail, pen-drive, file sharing sites). We chose this approach instead of a database to facilitate strategy sharing among our users.

The design tool provides the user with a graphical user interface (GUI) that is context sensitive and minimalist. The objective is always to minimize the amount of buttons being shown to the user and ensure that no button is shown out of context. This GUI works together with a click-to-select interface where the user can select which element he wishes to interact with (e.g., a SM, a ST, a player role). Every time the user selects an element through click the correct context menu appears in the GUI.

The design tool is divided into 4 main screens:

- Strategy file browser: The strategy browser is the initial screen and it shows a list of every strategy file in the user's device. In this screen the user can create, rename, delete and open each file. Once the user opens a strategy file he is taken to the strategy graph screen.
- Strategy graph editor: The strategy graph editor shows the strategy graph for the current TSM being edited. The user can select each SM and each ST on the graph and request to edit them. If the user requests to edit a SM he is taken to the strategy map editor and if he requests to edit a ST he is taken to the strategy transition editor.
- Strategy map editor: The map editor shows a single SM from the TSM and allows the user to edit the properties of the map. When he is done he is taken to the graph editor.
- Strategy transition editor: The transition editor shows a single ST from the TSM and allows the user to edit its actions. When he is done he is taken to the graph editor.

The remainder of this section will go into more detail about the 3 editor screens of the software. The file browser is fairly standard and requires no further detailing.

### 6.1.1 Strategy graph editor

The strategy graph editor is the main hub of the designer tool. From this editor the user can create new SM, ST and really shape the strategy graph to his will. The screen will show a graphical representation of the strategy graph and allow the user to move each node through drag-and-drop and select any node or edge through clicking. The information provided by the user in this editor allows the creation of a strategy graph as described in 3.2.3.

The commands the user can perform, other than dragging maps to reshape the graph, are conditioned to what is selected in the graph. There are 3 possibilities of selection:

- Nothing: When nothing is selected the user can create a new map in the strategy. The new map will have a default player role configuration and it will be placed in the middle of the screen with a standard name. The standard name is a simple numeric count that increments with each new map. This manner of naming new maps minimizes the amount of renaming necessary, even negating the necessity in some naming conventions.
- Node (SM): If a node in the graph is selected, the player can:

  - Create a new play: This automatically creates a new SM in the strategy and a new ST from the selected SM to the new SM.
  - Create a new connection: This requests that the user indicates a destination SM and creates a new ST from the selected SM to the indicated SM.
  - Edit SM: This opens the selected SM in the strategy map editor.
  - Pick color: This allows the user to pick any color for the selected SM to be represented in the strategy graph. It is sometimes useful for the designer to color SMs differently based on some semantic value. For instance, a common approach is to set terminal SMs as orange and central SMs as green to facilitate the visualization of the plays.
  - Rename: This allows the player to pick a name for the selected SM. We try to minimize the need of this command by giving each new SM a reasonable name, but sometimes the user might want to set custom names.
  - Delete: This command removes the SM from the TSM. It will also remove any ST that originates from or reaches the selected SM to keep the strategy graph consistent.
  - Set Reward: This allows the user to set the reward value for the selected SM. It is done through a horizontal slider that clamps the value between 0% and 100%.

- Edge (ST): If an edge is selected, which is done by clicking on the edge's name label, the user can:

  - Edit: This opens the selected ST in the strategy transition editor.
  - Pick color: This allows the user to set any color for the arrow that represents the ST in the graph. This is useful to highlight different plays, for instance, leaving riskier plays highlighted in red.
  - Rename: This allows the user to rename the transition, which will change the name shown in the transition label. The default name of a ST is $x - y$, where $x$ is the name of the origin SM and $y$ is the name of the destination SM.
  - Delete: This allows the user to remove the ST from the TSM.
  - Set Risk: This allows the user to set the risk value for the selected ST. It is done through a horizontal slider that clamps the value between 0% and 100%.

The strategy graph editor also allows the user to move the whole graph and change the zoom of the visualization. This enables an overall view of the strategy with a zoomed out option or a detailed strategy view with a zoomed in and dragging option. The graph editor also provides the user with important visual clues that facilitate the design of robust strategies. Each transition that is not consistent and requires some attention is drawn by a dashed line, instead of a continuous line. A transition where the ball ends with a player that is different from the player in the destination SM would be drawn as a dashed line, for example.

Figure 6.2 shows an example of the strategy graph editor. In the figure the SM with the number 3 is selected, as indicated by the green colored highlight. The currently set reward value for this SM is 0.66, as indicated by the value next to the slider at the bottom of the screen. We can also note that the ST $3 - 4$ requires the designer's attention, as it is represented by a dashed line. Another thing to notice is that there is more of the graph off to the bottom of the screen, as this is a zoomed in view of this portion of the strategy graph. The buttons on the right side represent the commands that the user can execute with this SM selected. The most used buttons in this context menu all have keyboard shortcuts indicated below their name. The items on the left side of the screen allow the user to test the TSM and inform the file to which this strategy is being saved. We will explain the testing tools later in this chapter.

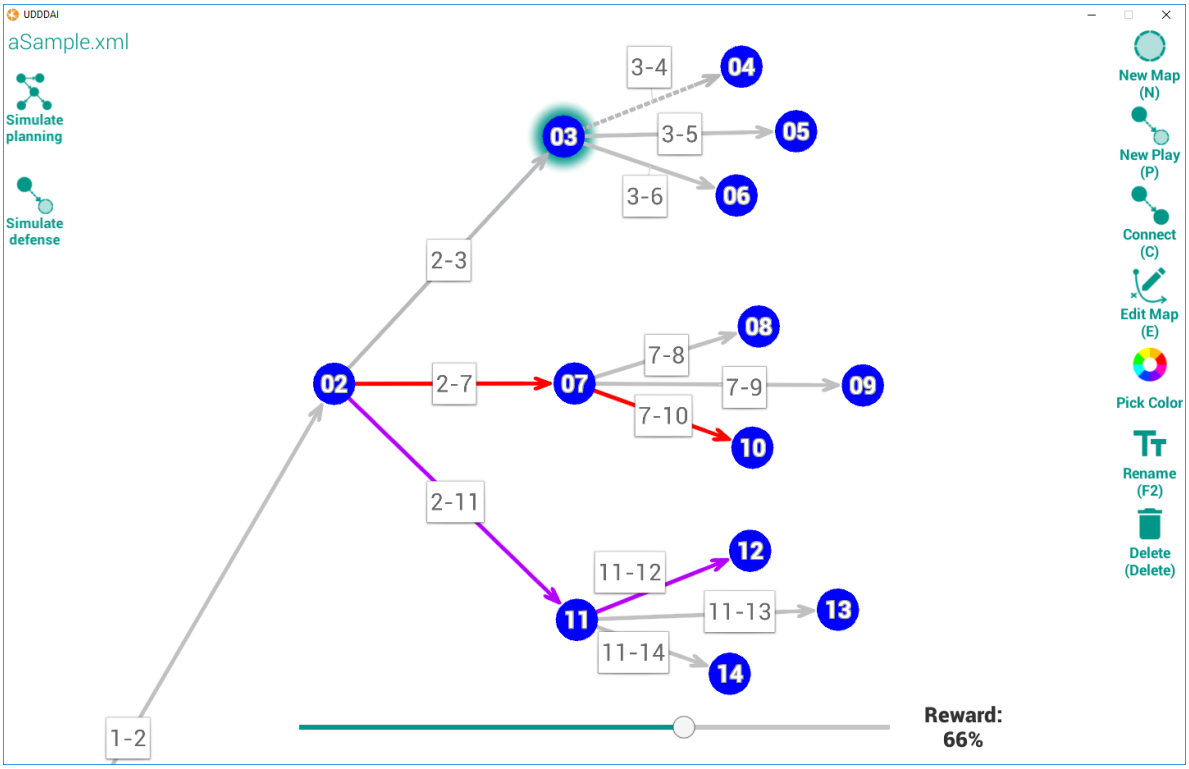


**Figure 6.2:** *The strategy graph editor of the basketball TSM designer.*

### 6.1.2 Strategy map editor

The strategy map editor allows the user to set the necessary information for each player role of the SM being edited. The information that the user needs to provide to this editor is enough to create a strategy map as described in 3.2.1. In this editor the user can move the center of each player role by drag-and-drop and he can select any player role by clicking.

The user has the following commands available, depending on what is selected:

- Nothing: When nothing is selected the user can go back to the strategy graph and also rotate the view. The default view of the court shows the user's team attacking from left to right while the rotated view shows the attack from bottom to top. The choice in camera rotation allows the designer to work with the representation they are more used to.
- Player role: When a player role is selected the user can:
  - Set ball: Set the player role as the ball holder.
  - Pick color: Allows the user to pick any color to represent the player in this SM. By default every offense player is represented by red and every defense is represented by blue.
  - Change rotation: A rotation icon that can be dragged and rotated appears and allows the user to rotate the ideal facing of the player role.
  - Change scale: Two sliders appear over the role's playing area and they allow the user to set the height and width of the playing area ellipsis for the selected player role.

If the user wishes to use group by omission on the information about a player role, they can simply leave the role on the outside of the court. The system will ignore the position of player roles outside the court when considering the equivalence class of the SM. It is worth noting that we always consider a default playing area for the defense players: each defense player role has a circular playing area of $1m$ diameter and the user cannot edit it. We decided to use this simplification because the defense players are mostly left out of the court and it would rarely be important to set different

playing areas for defense players. The defense players only need to be specified (i.e., put in the court) when there are two similar classes of situations in the strategy where the only difference is the positioning of the defense players. While we do support this kind of strategy, in our experience they are not very usual.

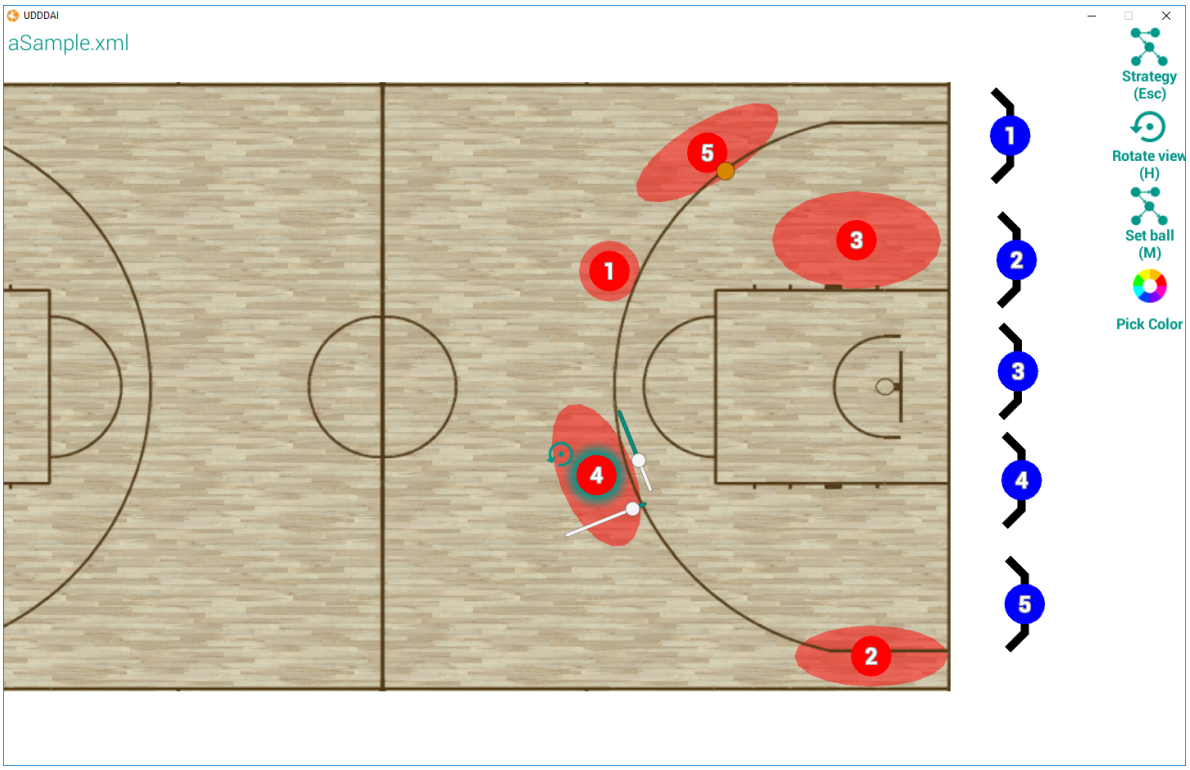


**Figure 6.3:** *The strategy graph editor of the basketball TSM designer, with a player role selected.*

Figure 6.3 shows an example of the map editor. In the figure the player role number 4 is selected, as indicated by the green highlight and the floating GUI around the player. We can also see the ball icon over player role 5, indicating that it has the ball possession. Note also the small circular icon above and to the left of the selected player role. This icon can be grabbed and rotated to set the rotation of the playing area. The two sliders for the size of the playing area rotate with the playing area to make it easy to see which slider controls which dimension.

### 6.1.3 Strategy transition editor

The strategy transition editor is by far the most complicated of the editors in the TSM designer. The reason is simple: STs have more information in them and the information is more dynamic (i.e., time is a factor in ST design). The information necessary to specify a ST is described in 3.2.2. A way to incorporate time information in the description of planning problems is explained in [FL03]. Our approach tries to facilitate the input of this information by providing the user with time specification tools.

Each player role in the strategy editor will have a path with at least 2 waypoints, the first located in the spot of the center of that role in the origin SM and the last in the same spot in the destination SM. The user can move each waypoint by drag-and-drop and he can create a new waypoint in the path by clicking on any position in the path. A newly created waypoint can be moved in the same way. If the user moves the first or the final waypoints he will also move that role in the corresponding SM. This facilitates the tweaking of strategies but must be used with care, as it generates a side effect. A pass can be created by right-clicking the path of the player with the ball and dragging the newly created pass arrow to the pass receiver's path.

The commands available to the user in the transition editor depend on what is currently selected:

- Nothing: When nothing is selected the user can go back to the strategy graph, rotate the view or switch the mode to REC or Set Passes. The view switch functions the same way as in the map editor, simply rotating the court from a left-to-right view to a bottom-to-top view. The REC mode will be explained in detail further on. The set passes mode is useful in devices without mouse (e.g., tablets), where right-clicking to create passes is impossible. When in set passes mode, the regular click (or touch) becomes a pass creation command and the user can no longer move waypoints (he needs to leave this mode to be able to move waypoints again).
- Waypoint: When a waypoint is selected the user has the option to set the time delay for that waypoint using a slider. A button to delete the waypoint also appears if the selected is not the first nor the last in the path.
- Pass: A selected pass gives the user the choice to delete it.

We created a time slider in the bottom of the transition editor window that allows the user to scrub through the transition to see the players moving. We did this to help visualization with the increased complexity of dealing with time as a variable. The user can drag the time slider to see any point in the transition, much like the seeker slider in most video players.

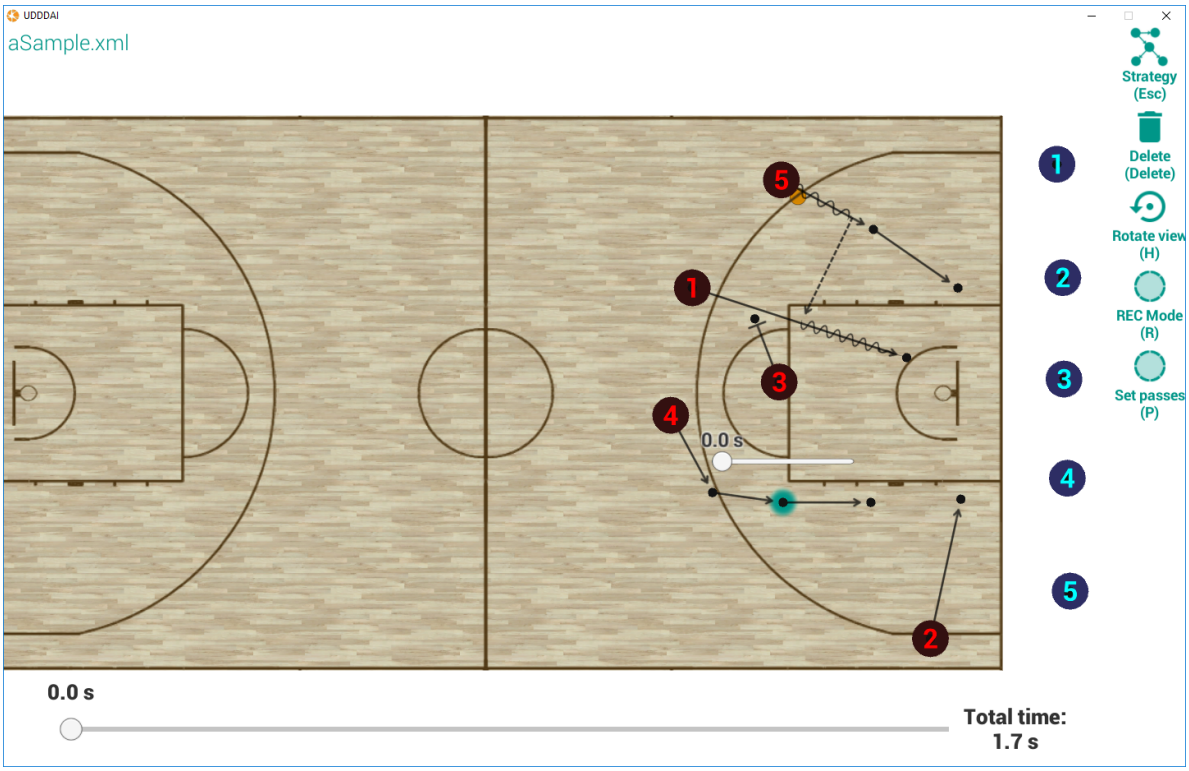


**Figure 6.4:** *The strategy transition editor of the basketball TSM designer, with a waypoint selected.*

Figure 6.4 shows an example of the transition editor. In this case the third waypoint in the path of player role 4 is selected, as indicated by the green highlight. The numbered circles indicate the position of each player role in that time of the transition. We can see at the bottom that the transition is 1.7 seconds long and it is currently showing the very start at second 0.0, hence all position indicators are on top of the first waypoint in each path. Note also that there is a pass near halfway through the transition from player role 5 to 1.

We can see that the squiggly line that indicates the carry action (i.e., moving with the ball) moves from one path to the other. This is done automatically to improve visualization, without any work for the user. Another automatic visual clue is the screen action feedback, that we can see being executed by player role 3 in the transition. A screen is automatically displayed when some waypoint in the screen executor path is put near the path of another player that passes the

waypoint while the executor is stationary there. The final clue that is put automatically by the software is the hand-over icon. If a pass is created in such a way that the distance traveled by the ball is very small (i.e., the sender and the receiver are near each other), then instead of a dashed arrow the software will place a # to indicate a hand-over of the ball.

The time slider was an important improvement to the initial version of the strategy designer. We learned that an immediate feedback for each transition that shows the synchronicity of movements is essential to a proper strategy design. Figure 6.5 shows an example of the use of the slider. Note how between the first and the second frame the user can verify if the timing of the screen performed by player role 3 is correct. Note also in the second frame that the ball actually travels through the path of the pass before reaching a pass receiver.

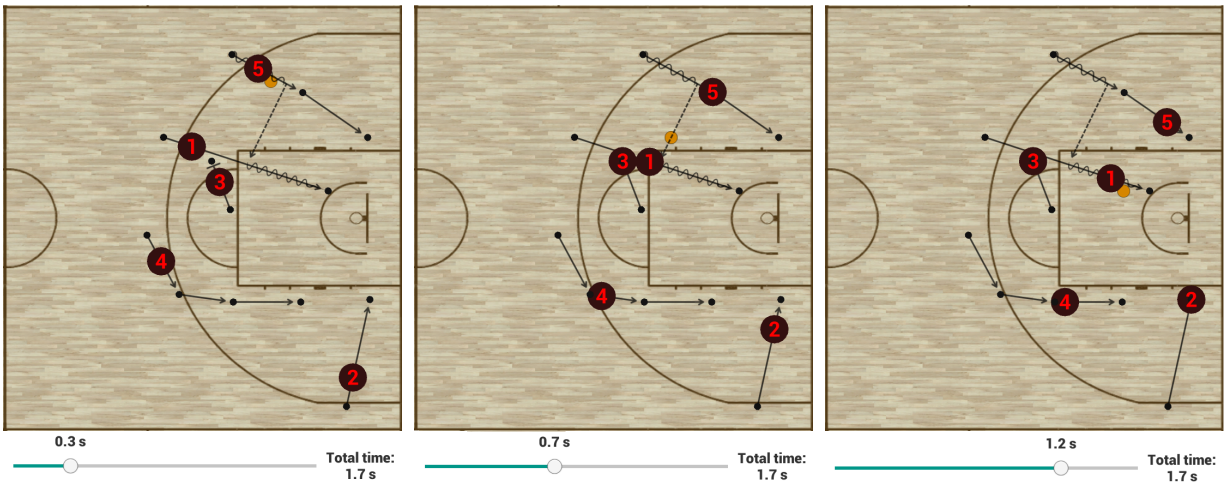


**Figure 6.5:** *Example of the use of the slider to scrub through a transition to check action synchronization.*

While it is possible to specify a transition by creating and placing every single waypoint, then setting the wait times and finally assigning passes, it is a tedious and long task. We created a REC mode to help the user specify most of the required information intuitively and quickly. The REC mode allows the user to draw the movement of each player on the screen, including the timing and positions of every waypoint. To do this the user simply needs to enter the REC mode and drag a player position indicator along the path they wish to set for that player role. While an indicator is being dragged they will automatically create the necessary waypoints to represent the path along the drag. To set a wait time the user simply needs to hold a position for the desired amount of time. Finally, when an indicator is being dragged the rest of the team will move along their current paths to show the user exactly what will happen in the transition while they are still creating it.

Figure 6.6 shows an example of the REC mode in 3 frames of a single drag of player role 4. Note how the other players move along their paths as the drag happens and the time slider is also updated automatically. The middle frame shows the visual clue that indicates that a wait time is being recorded. While the indicator is held in place the rest of the transition continues normally, indicating exactly the position they will be at during the wait period of player number 4. Finally, note that when the drag follows some curve, as in this example, multiple waypoints are created to accommodate the path smoothly. Using the REC mode, an experienced designer can specify complex transitions in under a minute.

## 6.2 Specifying the Basketball Defense's TSM

As specified in 3.3, the defense's TSM is much simpler than the offensive TSM due to its reactive behavior and large equivalence classes. Another simplification is that each defensive player behavior can be specified by looking only at the ball holder and the defender's marking target. So, the equivalence class of a SM in the defense's TSM is a simple composition of the equivalence classes of each of the 5 defenders.

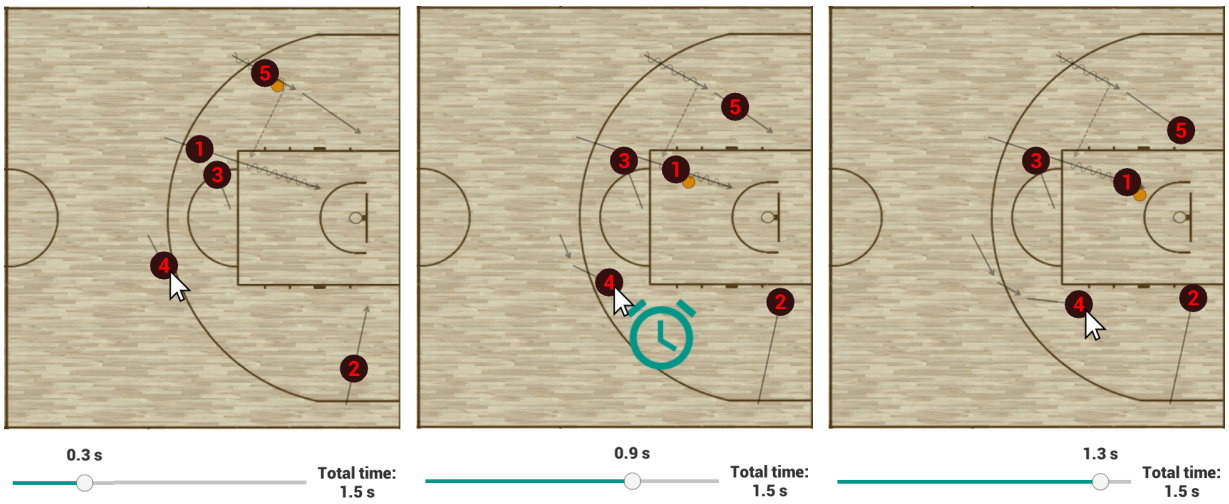


**Figure 6.6:** *Interface REC onde o usuário define a movimentação do jogador número 4 enquanto o resto da transição ocorre automaticamente a sua volta.*

To simplify the model even further, we also assume that each defensive player will have the same behavior when in the same situation. This final simplification allows the implementation of the defense's TSM designer with a simple table. In this table we create a row for each possible situation equivalence class for a single defender and allow the user to fill the row with the desired behavior in that situation. As described in 3.3.1, each behavior can be specified by setting 3 parameters each with at most 5 possible values. We facilitated the input of a TSM by creating simple drop-down lists for each parameter.

| Header | Explanation |
|---|---|
| Ball Area | Position of the ball holder |
| Target Area | Position of the target player |
| Side | Target on same or opposite side of ball |
| Direction | Direction from target for marker |
| Distance | Distance from target for marker |
| Gaze | Marker gaze direction |

| Possible Target Areas | Explanation |
|---|---|
| TK | Top Key |
| W | Wing (both L and R) |
| C | Corner (both L and R) |
| STK | Short top key |
| SW | Short wing (both L and R) |
| SC | Short corner (both L and R) |
| PC | Paint center |
| PS | Paint side (both L and R) |
| BALL | Target is ball holder |
| HOME | Target on his home court |

| Possible Directions | Explanation |
|---|---|
| cover basket | Between target and basket |
| cover ball | Between target and ball |
| cover center | Between target and court center |
| along paint | Paralel to target along paint line |
| along 3 pts | Between target and basket along 3pt line |
| ... | |

| Possible Distances | Explanation |
|---|---|
| min (0.5m) | Minumum mark dist, touching, 0.5m |
| med (1m) | Medium mark dist, cover pass/shoot, 1m |
| large (2m) | Large mark dist, cover movement, 2m |
| none (0m) | special case (for inside paint, etc) |

| Possible Gazes | Explanation |
|---|---|
| look target | Look at own target |
| look ball | Look at ball holder |
| look sideline | Look at sideline (arms paralel to sideline) |
| look endline | Back to basket, looking forward |
| look both | Look between ball and target |

| Ball Area | Target Area | Side | Direction | Distance | Gaze |
|---|---|---|---|---|---|
| TK | BALL | - | cover basket | none (0m) | look target |
| W | BALL | - | cover basket | none (0m) | look target |
| C | BALL | - | cover basket | none (0m) | look sideline |
| STK | BALL | - | cover basket | none (0m) | look target |
| SW | BALL | - | cover basket | none (0m) | |
| SC | BALL | - | cover basket | none (0m) | |
| PC | BALL | - | cover basket | none (0m) | |
| PS | BALL | - | cover center | none (0m) | |
| HOME | BALL | - | along 3 pts | large (2m) | |
| TK | TK, W | SAME | cover basket | med (1m) | |
| TK | TK, W | OPPOSITE | cover basket | med (1m) | look sideline |
| TK | C | SAME | along paint | med (1m) | look ball |

look target
look ball
look sideline
look endline
look both

**Figure 6.7:** *The table used by the defense TSM designer.*

Figure 6.7 shows an example of the table created for the designer. The 5 small tables on the top of the sheet are simply explanations for the codes that appear on the main table (on the bottom of the figure, shown cutted after the first few rows). The first two columns of the main table specify the area of the court where the player with the ball and the marking target are, respectively. The possible values for these fields are shown in figure 3.10. The column labeled "Side" indicates whether or not the ball and the target are in the same side of the court (with values "SAME" or

"OPPOSITE"). This allows for a more compact table due to the symmetry assumption explained in 3.3.1. The job of the user is to specify the values for the last 3 rows of the main table: Direction, Distance and Gaze. The combination of these values completely specify the behavior of a defender. In the figure we can see that the value of cell 27F is being specified by choosing a value from the drop-down list. The entire main table has 108 lines.

A completed table generates a simple file that can be read by the system and used during the simulation to guide the defensive team behavior. The first step to control the defense is to allocate a defender for each offense player. There are 3 approaches to allocate the defense, which are analogous to the allocation methods described in 4.2.1:

- Fixed: The allocation is decided at the beginning of the match and maintained throughout the simulation.
- Greedy: We allocate the closest defender and offensive player pair to each other and proceed like this until every defender is allocated.
- Minimal: We choose the allocation that minimizes the sum of all distances between offensive player and designated defender.

Once the allocation of defenders is complete, the system matches the situation of each defender to a row in the table. The parameters in the row will be used to specify a position and rotation for the defender. It is not easy to visualize the behavior of the defense from the rows in the table, so we also created a section in the simulation software that allows the user to see the defense reacting to any given match situation. Figure 6.8 shows the screen of the software that is used to test the defensive behavior. In this section the user can freely move the offensive players by drag-and-drop and the software will automatically place the defenders in their ideal positions and rotations. The offensive players are annotated with the area of the court where they are located to help the user identify which equivalence class each defender is currently in.

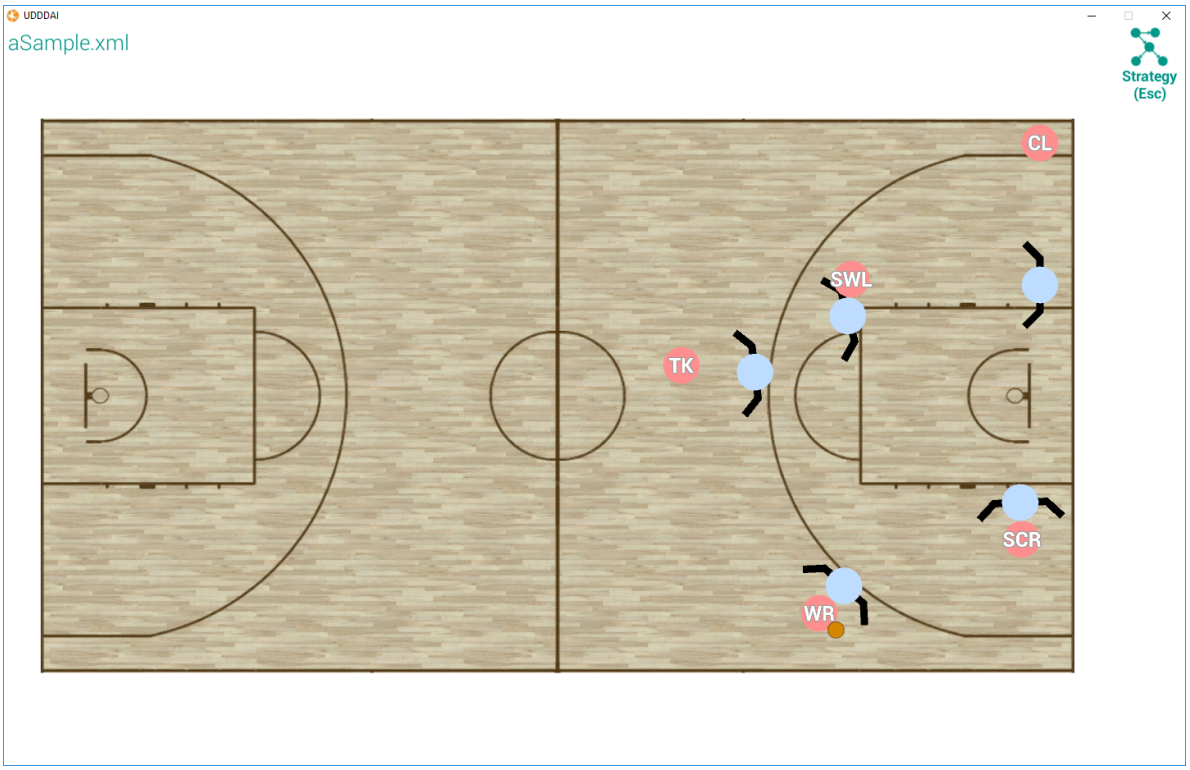


**Figure 6.8:** *The defense's TSM tester screen.*

## 6.3 Implementing the offense collective layer

### 6.3.1 The Matching Implementation

The basketball's matching process is described in detail in 4.2.1. In our implementation we need to consider 6 possible combinations of type (temporal or spacial) and allocation method (fixed, greedy and minimal). We developed a system that leaves the choice up to the user, since we believe each combination has their own benefits and drawbacks.

The matching algorithm depends heavily on the implementation of the *p-dist*, so we will start by that function. The distance from ellipsis to point is not trivially implemented, but lucky we can get an excellent approximation very quickly. The first thing to note is that the closest point will be on the edge of the ellipsis whenever the point is not inside the ellipsis. The second important thing to note is that the intersection between the perimeter of the ellipsis and the line segment defined by the point and the center of the ellipsis is not the nearest point, but it is a decent approximation, as shown in figure 6.9. The idea of the algorithm is to improve the approximation of the the point $x$ by rotating a seeker line. We initially define a seeker line as a line segment that connects the center of the ellipsis and the base point $r$. Finally, we rotate the seeker line in both directions and check which direction improves the approximation. The algorithm is essentially:

To find the closest point from $p$ that is in ellipsis $E$:

1. Define the point $E.center$ as the ellipsis center and vector $E.forward$ as the positive main axis of $E$.
2. Define the function $E.border : \mathbb{R} \rightarrow \mathbb{R}^2$ so that $E.border(\beta) = q$ means that the angle between $(q - E.center)$ and $E.forward$ is $\beta$.
3. Find $\alpha$, the angle between $(p - E.center)$ and $E.forward$.
4. Find $r = E.border(\alpha)$ .
5. Define $\delta$ to be the search increment, initially $10°$.
6. Search for a better $\delta$-approximation:
   (a) Find $r^- = E.border(\alpha - \delta)$ and $r^+ = E.border(\alpha + \delta)$.
   (b) Find the smallest distance among $dist(r^-, p)$, $dist(r^+, p)$, $dist(r, p)$:
   If it is $dist(r^-, p)$, define $r = r^-$ and $\alpha = \alpha - \delta$. Repeat step (a).
   If it is $dist(r^+, p)$, define $r = r^+$ and $\alpha = \alpha - \delta$. Repeat step (a).
   If it is $dist(r, p)$, $r$ is the $\delta$-approximation. Decide whether to decrease $\delta$ and repeat step 6 or stop here and return $r$.

One obvious performance optimization when coding the above algorithm is to verify which angle created the smallest distance (between $r^+$ and $r^-$) and use this information to avoid recalculating the $E.border$ for that angle in the next repetition of step (a). In order to define the function $E.border$ we will need the values for the width and height of the ellipsis. The width is the distance from center to border along $E.forward$ and the height is the distance from center to border along the orthogonal axis to $E.forward$, which we will call $E.up$. The width and height values are shown in figure 3.8 and we shall refer to them as $E.w$ and $E.h$ respectively. If we guarantee that both $E.forward$ and $E.up$ are normalized vectors, then we can define $E.border$:

$$E.border(\alpha) = E.center + (E.forward\ width\ \sin(\alpha)) + (E.up\ height\ \cos(\alpha)) \tag{6.1}$$

In order to calculate the m-dist we need to use the p-dist implementation and consider each player allocation method. There are 3 different approaches, depending on the allocation method chosen:

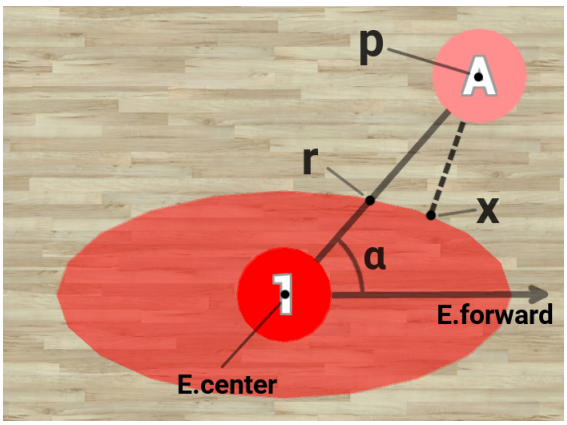


**Figure 6.9:** *The initial step in the algorithm to calculate the p-dist. The point we seek is $x$ and the initial approximation for the search is $r$.*

1. Fixed: To calculate the m-dist using a fixed allocation we calculate the p-dist between each player and their respective roles in the fixed allocation. Then the m-dist is the sum of all p-dist in spacial matching or the largest p-dist in temporal matching.

2. Greedy: Find the ideal allocation as follows:

   (a) Start with a complete list of players and player roles.

   (b) Find the closest pair of player and player role in the list, using p-dist. Then assign the player to the role and remove both from the list.

   (c) Repeat step (b) until the list is empty.

   Using the allocation found, the m-dist can be calculated in the same manner as with the fixed allocation.

3. Minimal: For each possible allocation calculate the m-dist the same way it was calculated in the fixed allocation approach. Then pick the allocation with the smallest m-dist and use that as the value of m-dist for the SM.

Since we only calculate m-dist when we are executing a matching function, we know that we will always calculate the m-dist for every SM in the TSM. This allows for a very simple optimization: we can keep the best m-dist found and use it to cut some calculations short. The exact method used to optimize depends on the type of matching being done:

- Temporal: When executing a temporal matching we can stop the calculation of a new m-dist if we find a p-dist that is larger than the best m-dist found.

- Spacial: When executing a spacial matching we can stop the calculation of a new m-dist if the sum of the p-dist reaches the value of the best m-dist found.

So the matching function will calculate the m-dist for every SM and select the one that minimizes the m-dist. Additionally, the function finds the appropriate allocation of players to roles. We have implemented many methods of finding the match, each with their advantages and drawbacks. Since it is not trivial for the user to visualize exactly how each method behaves in the simulation, we created a matching test environment. In this screen the user can position the players on the court and see what would be the result of each matching method in their TSM. Figure 6.10 shows an example of the matching tester. The user can move each player by drag-and-dropping and the buttons on the right allow them to select the matching method. The toggle in the bottom of the menu switches between temporal and spacial matching and each of the other buttons executes the matching with a different allocation approach. Note that each player outside the playing area of their role is linked to that area by a dashed line, which helps the user understand what allocation was used.

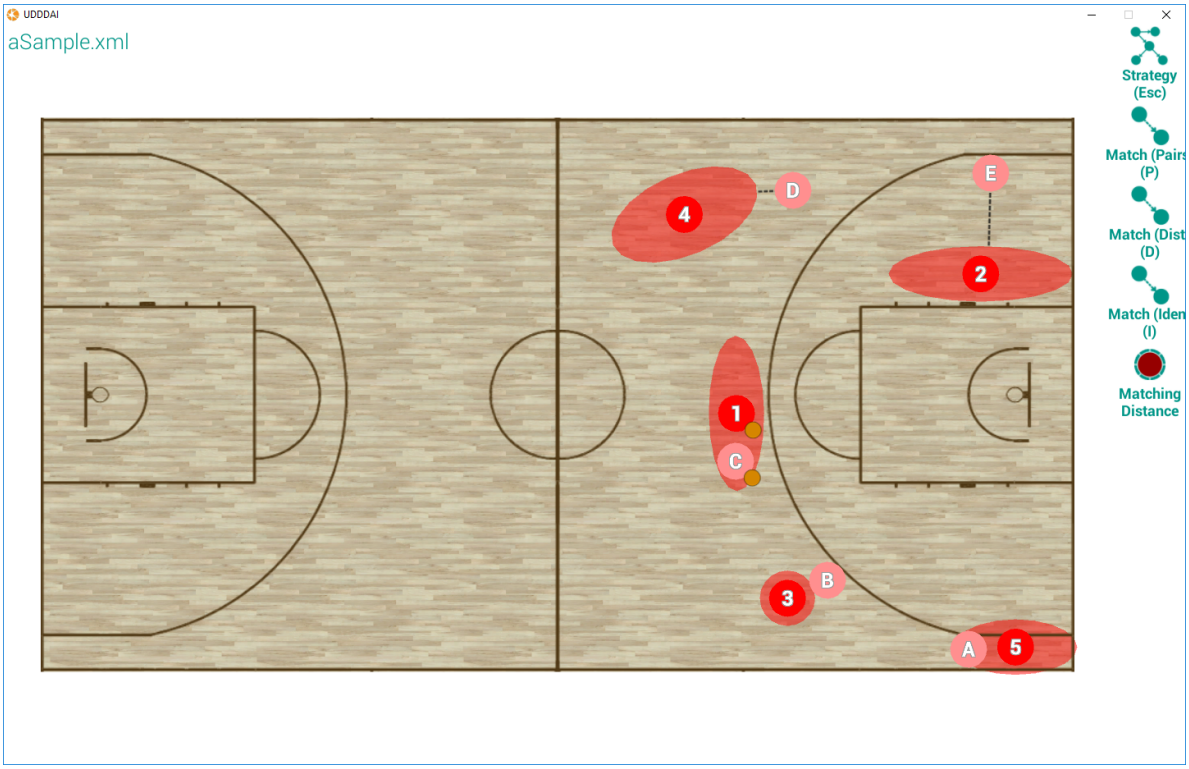


**Figure 6.10:** *The player allocation and matching tester screen.*

### 6.3.2 The Planning Implementation

The basketball's planning process is described in detail in 4.2.2. In our implementation we needed to consider the two approaches to planning:

1. Short-term planning:
   (a) Add the initial SM (found by the matching function) to the play.
   (b) Create an empty candidates list. Add to the list every ST that:
      - Originates on the last SM added to the play.
      - Has a destination SM that is not yet on the play.
   (c) If the candidates list is empty, the play is complete. Else, for each candidate in the list, calculate the score as $(1 - ST.risk)\ ST.destination.reward$.
   (d) Randomly select one of the candidates using the method described in 4.1.2 and add it to the list. Repeat from (b).
2. Long-term planning:
   (a) Create an empty list of completed plays
   (b) Create a list of partial plays, add the initial SM to the partial plays list.
   (c) While the partial plays list is not empty:
      - Remove some play from the partial list, call it $P$.
      - Find every ST that can be added to the play without creating a loop.
      - For every ST found insert a new play in the partial list. The new play is $P$ with ST added to the end.
      - If no STs were found add $P$ to the completed list, else discard $P$.
   (d) Score every play in the completed plays list by multiplying $(1-ST.risk)$ and $ST.destination.reward$ for every ST in the play.

(e) Randomly select one of the plays using the method described in 4.1.2.

As with the matching function, we also created a test environment for the planning. In the planning tester we allow the user to set the position for every player and then request that a play be planned. The tester then shows a graphical representation of every action in every ST of the planned play. A time slider is also created and it allows the user to scrub through the whole play to check synchronicity. The user can repeatedly request new plays and for each request the tester will randomize a play that fits following the algorithm described above. Figure 6.11 shows the interface of the planning tester. Note that the buttons on the right side allow the user to specify the type of matching and the type of planning he wishes to be used.

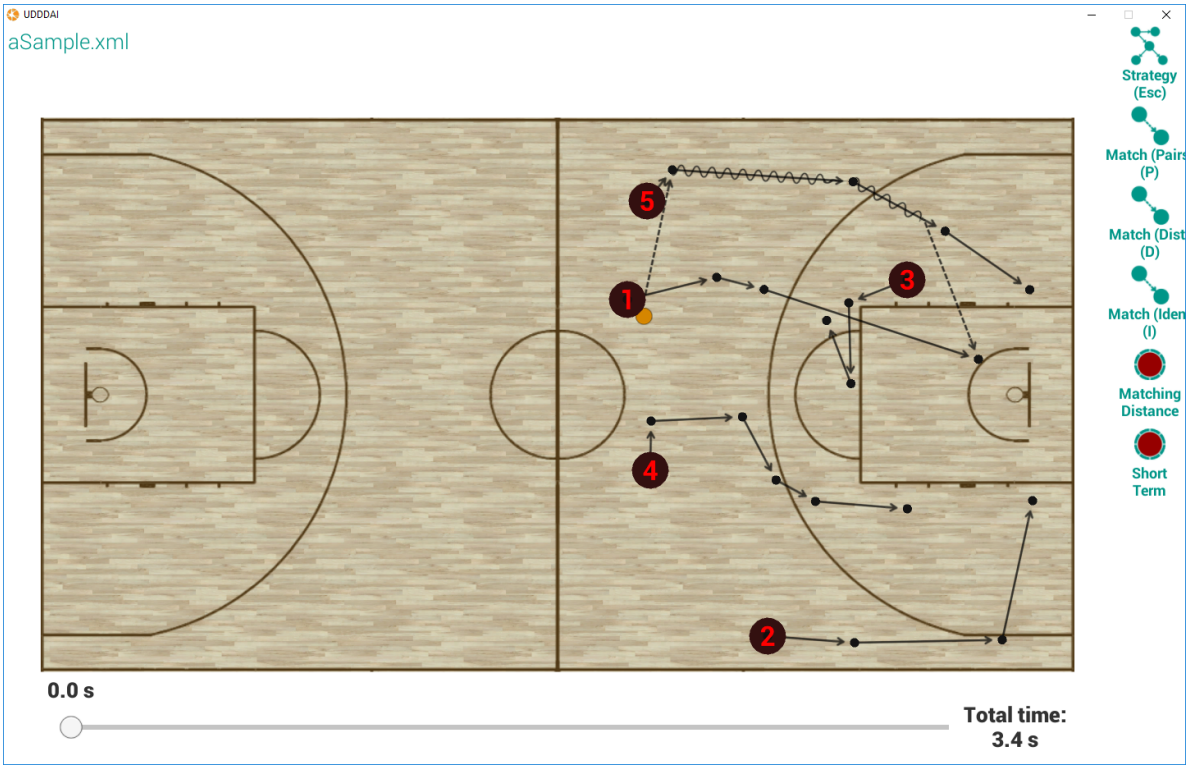


**Figure 6.11:** *The planning tester screen with a play being displayed for some player disposition.*

### 6.3.3 The Realization Implementation

The realization function can be implemented in many ways, as we have described in 4.2.3. We initially decided to implement the centralizer approach due to its simplicity and the fact that the plays are more easily recognized when using this approach. We create a move action for each player that takes them to the center of their respective playing areas and a wait action to ensure that the execution of the play will be started by every player at the same time. The realization function also creates a pass action to be executed during the move if the player holding the ball is not the one that starts the play with the ball. We can see this in figure 6.11, where player 1 passes the ball to player 5 as he moves to the initial play position.

## 6.4 Implementing the individual control and physics simulator

We have implemented the individual layer described in 5.2.1 by creating a simple control structure that already integrates with the physics simulator described in 5.3. There is a simulator controller that holds two teams, each with 5 players. Each player is responsible for their own actions. The root of the control structure is called the *Simulation Manager* and it is responsible for ticking the simulation each frame as well as triggering the collective planner.

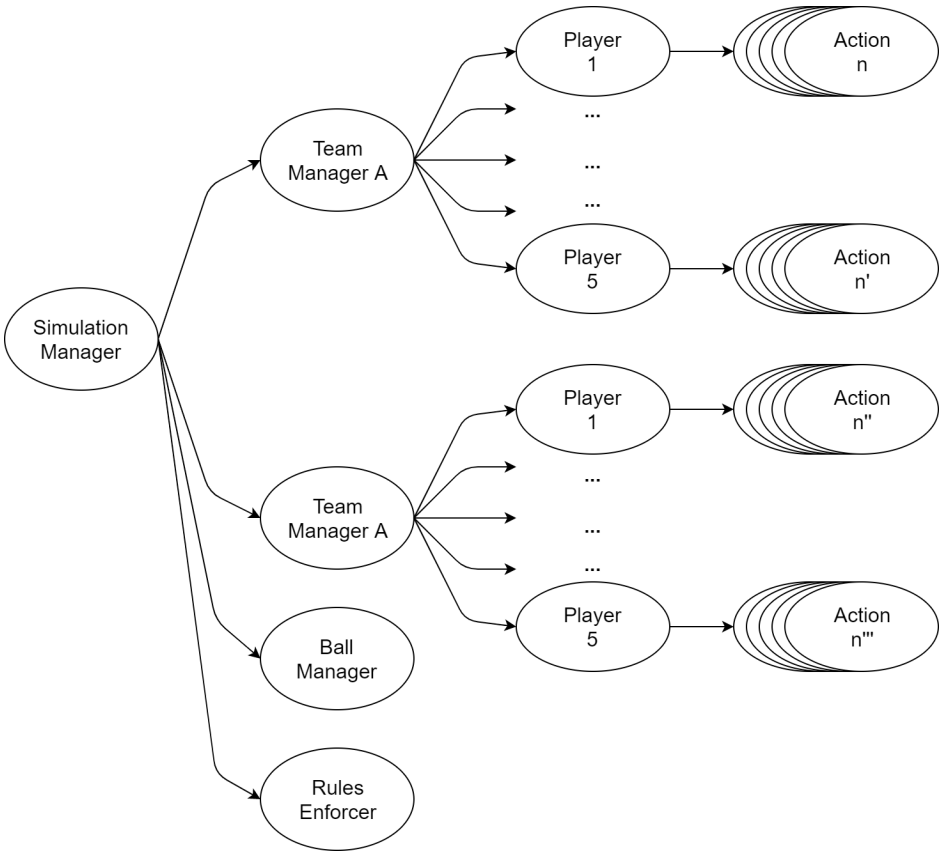


**Figure 6.12:** *The control structure for the basketball simulator.*

Figure 6.12 illustrates the control structure of the simulator. When a collective planning needs to occur, the Simulation Manager triggers it and the resulting collective play is sent down the structure until it reaches the players. Each player then creates the necessary actions and adds them to their own queue. This process will be triggered again every time a new plan is necessary. This happens when an action fails in some player queue or when a player queue becomes empty. The collective plan adaptation process described in 5.2 actually occurs in each of the players. The actions created by each player in their queue are instances of one of the following actions: Move, Wait, Pass, Receive, Shoot and Steal.

The Simulation Manager centralizes the passage of time during the simulation. This greatly reduces the effort to create effects such as freeze/pause time or control the animation time scale (e.g., slow motion, fast forward). During the simulation a time amount $\delta$ is decided before triggering a new simulation frame based on the time scale of the simulation. This $\delta$ is passed in a tick command to each team, then to the ball and finally to the rules checker. This sequence is the same that was described in 5.3, considering that the ball is the only environment executor in basketball. When each team receives a tick with a $\delta$ parameter it passes this down to each of their players. When a player receives the command he will execute actions as follows:

1. Execute the first action in the queue for $\delta$ seconds. Check the action return value:
   (a) If *running*, terminate this tick.
   (b) If *fail*, trigger new collective planning, terminate tick.
   (c) If *success*:
   - Get unused time $t$ from the return and set $\delta = \delta - t$.
   - Remove the action from the queue.
   - Go back to 1. and execute again.
2. If no actions remain in the queue, trigger collective planning.

A player will spend the whole time it has available executing actions from his queue. The moment an action fails or the queue runs out of actions he will trigger a new collective plan. If an action succeeds he moves on to the next. If an action reports that is still running after executing the remainder of the player's time he will terminate that tick.

Each action execution will update the match situation to reflect its effects (e.g., move the player). Actions that require some form of statistical simulation, such as ball disputes, will be registered in a judge that will decide the result based on the model of the players involved. The tick sent by the Simulation Manager will reach the ball after every player has been updated and finally the rules enforcer will update and ensure that every rule is being followed (e.g., the ball is not out of bounds). If some rule requires a break in the simulation, such as a point being scored, the enforcer will send this information to the Simulation Manager and trigger a re-plan.

## 6.5 Implementing the mechanical control

The mechanical control layer of our simulator needs to handle the body movement of each player and visual clues regarding the speed of the simulation and the score. We also implemented a system to control the camera position to enhance the user's experience while watching the simulation. None of the effects of this layer directly affect the simulation, but they each contribute to the overall quality perceived by the user. We divided the mechanical control into 3 parts: animation controller, camera control and user interface.

### 6.5.1 Animation Controller

The animation controller uses information from the simulation to manage the animation blend and inverse kinematics (IK) for each player in the simulation. The objective is to correctly display the players' body motion even though we have a limited number of animation clips. Figure 6.13 shows an example of how this can be achieved with animation blending. The first and last frames of the figure are created with preexisting animations and they represent movement speeds of $1.1m/s$ and $2.4m/s$, respectively. The middle left and middle right frames were achieved by blending different proportions of the preexisting clips to achieve animations for speeds of $1.5m/s$ and $2.0m/s$, respectively. The windows in each frame in the figure show a graphical representation of the blends. The center of the each window represents the stationary position. The $y$ axis reflects the forward/backward speeds and the $x$ axis represent lateral movement. The blue dots are preexisting animation clips and they are placed in the window according to their represented speeds. The red dot going up as the frames progress is the current movement speed. We can see that a circle around the blue dots that represents how much of that clip is being used to create the blend. As the speed increases the proportion of the walk clip used diminishes as the proportion of the run clip increases.

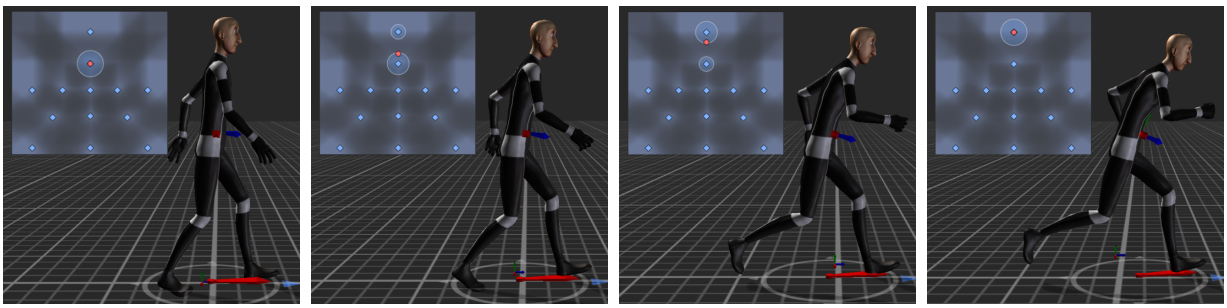

**Figure 6.13:** *The blending of walk and run forward animations. The animations in the middle frames are achieved through blending of the left and right frame animations.*

We created a layered finite state machine to control all the animation blends used during simulation. Each layer that is added on top of the base layer is used to override some parts of the

player's body. For instance, we created a layer above the base layer that overrides the animations of the arms when the player has the ball. The state transitions in this layered FSM are triggered when relevant simulation events occur. We use the following simulation events to control animation blending:

1. Simulation time scale: This is used to handle the playback animation speed. The default playback speed is multiplied by the simulation time scale to achieve a coherent effect. For instance, if the simulation plays at half the speed, the animations will also play in slow motion, at half speed.

2. Player speed: How fast the player is moving in the simulation is used to control the blend between idle, walk and run animations.

3. Player direction: The direction of the player, relative to the speed, is used to control the blend between front, side and back facing animations.

4. Ball possession and actions: The player with the ball (if any) will use different animations to represent the ball domain. Ball actions trigger different animations to be played once, such as a shooting ball animation.

5. Ball position: The ball position, when it is not in the possession of a player, is important to guide IK chains of players moving to get the ball. We define IK chains in the arms of the player and ensure that he moves his hands towards the exact position of the ball.

6. Defense players' targets: The marking target is used to control head IK chains to make the defender look at the target and also to set his animations to marking stances.

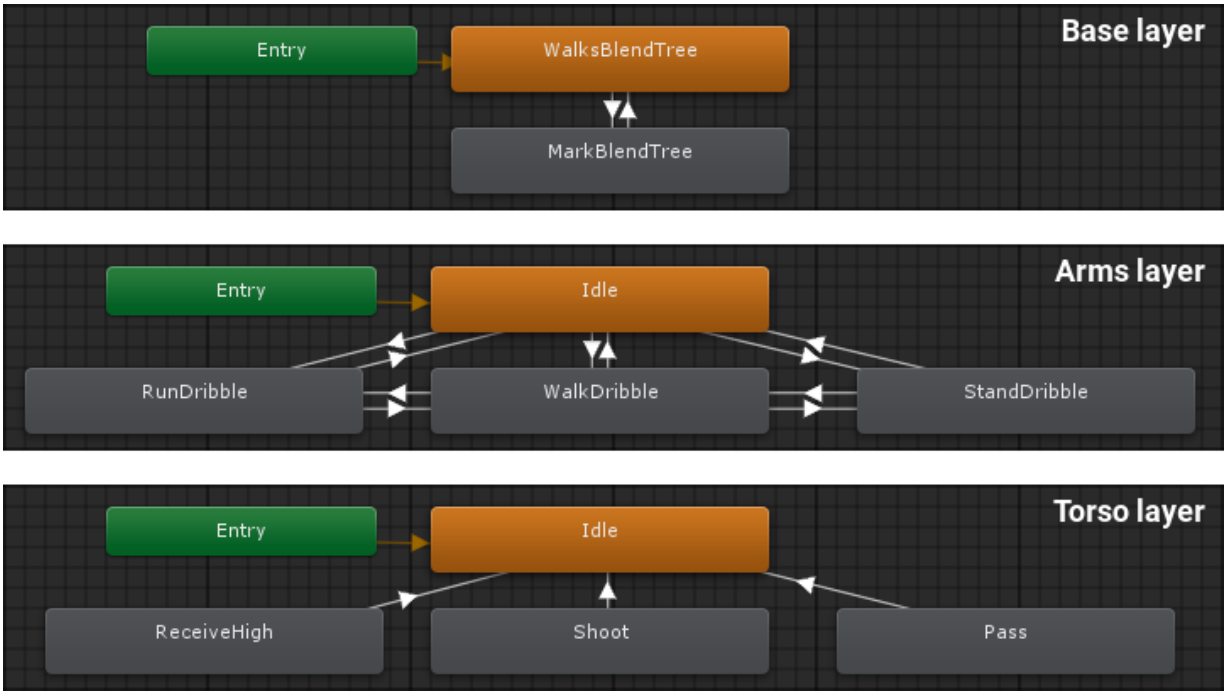


**Figure 6.14:** *The layered finite state machine to control player animation during the match simulation.*

Figure 6.14 shows each of the 3 layers of the FSM we created to control each player during the match simulation. The purpose of each layer and the animations contained in the states are:

1. Base Layer: This is the layer that controls the main portion of the players' bodies. The layer has two states:

   (a) The Move Blend: This animation blend state has an idle clip and clips for walk and run in all 4 directions (back, forward, right and left). It has 13 animation clips in total and the exact clip blend is selected by the decomposition of the speed in two axis: one parallel to the players facing direction and one orthogonal to it.
   (b) The Mark Blend: This animation blend state works exactly the same way as the move blend, but some of the animations are swapped to mark animations. A mark animation puts the player's arms out stretched and lowers his stance.

   The transition between each blend state verifies if the player is marking someone or not.

2. Arms Layer: The arm layer overrides the animations for the arms of the players. It is only used when a player has the ball, to change how he positions his arms while carrying the ball. This layer has 4 states:

   (a) Idle: This state does nothing and the arm animations default to the base layer. This state is used by any player without the ball.
   (b) Stand Dribble: This state has an animation clip that controls the arms in a continuous up and down motion. It is meant to override the arms when the player has the ball and is stationary, so the movement is relaxed and no swinging due to movement is present.
   (c) Walk Dribble: This state is just like the Stand Dribble, but it incorporates some arm swinging due to walking with the up and down motion that results from handling the ball.
   (d) Run Dribble: Same as Walk Dribble, but with even stronger arm swinging due to running.

   The transitions between Idle and the other states are conditioned by ball possession and the transition among the dribble states are conditioned by movement speed.

3. Torso Layer: This layer overrides the movement of the entire upper body. It is only used when a player is executing a pass, shooting or receiving the ball. It has 4 states:

   (a) Idle: This state does nothing and the upper body animations default to the base or arms layers.
   (b) Pass: This plays a pass animation clip where the player grabs the ball with both hands and does a passing motion.
   (c) Shoot: The same as the pass state, but with a shooting motion.
   (d) Receive: This state plays an animation clip where the player moves both hands up to chest area, catches a ball and holds it.

We have IK chains on the arms of the players that are used to smoothly transition the ball from a physics controlled state to an animation controlled state. The ball is controlled by the physics engine during moments where no player has possession. Once a player takes possession of the ball, the position of the ball simply follows an animated empty object on the player's model. This is done because it would be very difficult to simulate the player dribbling the ball and the end result would be essentially the same. We use the IK chain to edit arms animations in 3 cases:

1. When a player will take possession of the ball, an IK chain will guide his arms towards the direction of the ball and allow him to smoothly receive and transition it to an animation controlled state.

2. When executing a pass, an IK chain will rotate the body and arms in the direction where the pass receiver is, creating a realistic and continuous motion when the ball transitions back to the physics controlled state.

3. When dribbling with the ball, an IK chain will ensure the hands are touching the ball at the appropriate times during the animation. Figure 6.15 show an example of this IK chain in use. Note that the hand misses the ball when the IK is not on. This issue appears due to the player's body inclination not being accounted for in the arm animation, hence the need of an IK chain.


**Figure 6.15:** *The IK adjustment effect on the animation of a player conducting the ball while running.*

### 6.5.2 Camera control and user interface

Camera control and user interface are the means through which the user can manipulate their visualization of the simulation. Using the camera control they can set their preferred view angles and the user interface allows them to set the simulation speed, pause or play it.

We chose the camera angle options based on common broadcast angles of sporting events and video game industry standards. The options available to the user are:

1. Fixed points: This options allows the user to navigate between 8 fixed camera points around the court. There is one point in each corner and one in the middle of each court boundary line.

2. Sideline: The camera automatically follows the ball along the sideline. The user can control the height and distance to the ball.

3. Hover ball: The camera automatically follows the ball and the user can rotate the camera around the ball freely.

The user can also use the interface created by the mechanical layer to control the simulation speed. He can play and pause the simulation or set the speed anywhere from very slow motion (25% of regular speed) to very fast motion (16x regular speed).

## 6.6 Match simulator results

The basketball simulator was conceived to receive the complete strategy for both teams (offense and defense) as well as a set of player models to be used during the simulation of probabilistic actions, such as score attempts. At this moment, we have implemented the software to receive both teams' offense TSM and everything else was provided by a basketball specialist and integrated into the system. This approach simplified the end user experience and allowed us to focus the development on the offensive strategy. We concluded that the current defense specification method

described in 6.2 is more suited to a specialist on our team than the end user. We consider the possibility of creating a more intuitive input method for the defensive TSM in the future work section (7). Another simplification we currently employ is that every player model is the same and we do not require the user to provide it. This means we consider every player in the simulation to be identical, a fact reflected by our choice of robot avatars for each player. We also discuss the possibility of adding more diversity to the simulation through different player models in the future work section.

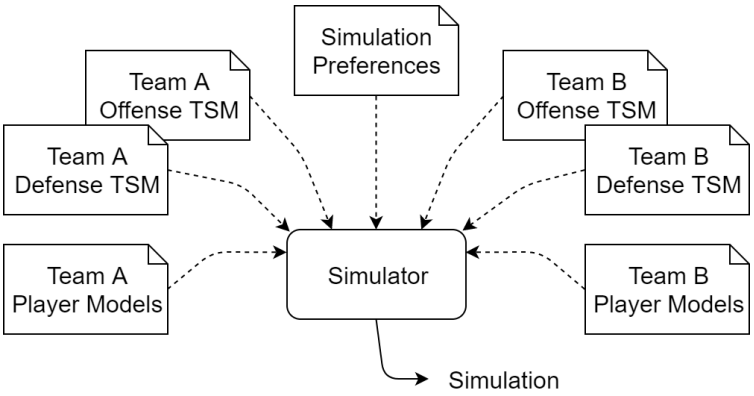


**Figure 6.16:** *The simulator input data, potentially (but not necessarily) provided by the end user.*

The user can create offensive TSMs using the design tool explained in 6.1. Once the simulator starts, the user can specify which TSM each team should follow. The simulation preferences are another important input for the simulation, specially in the early development stages when we wish to really understand the implications of each choice. The impact of the different preference choices may not be immediately obvious, but over time we believe they will become more evident.

The most striking result of implementing the basketball simulator was how quickly our methods allowed us to reach the current state the simulator. We managed to assemble every important module in a coherent and robust structure without sacrificing the ability to expand to new levels of sophistication. Each decision to simplify the model or remove features was taken consciously and we left the necessary hook to be able to simply introduce it at a later date. Using the formal model and designed architecture as guides, we constructed an empty skeleton of the whole system and then incrementally filled in the parts we judged most important. Using this method a single programmer managed to code the entire software solution without any major hurdles. While still far from complete, as evidenced by the suggested future work in section 7, the current version of the software already proves that the concepts presented in this thesis do in fact provide a strong basis for simulator development.

The modularity achieved due to the layered system and the separation between each level of control allows the developers to focus their work on the aspects judged more important. A developer can decide to focus efforts into making a single layer more sophisticated without needing to change the other layers. For instance, if one wishes to create a video-game where the user controls some of the players, they might want to focus more closely on the physical and mechanical layers. Doing so will achieve a higher graphical fidelity and by using the provided planners in conjunction with real time user input one can deal with players' action definition. Analogously, if one is simply interested in the nuances of strategic movement behavior, the physical and mechanical layers already provide the necessary base and the work can be focused on the planner layers.

We developed each control layer as much as time allowed aiming to prove that it is possible to create a very sophisticated system more easily using our methods. We consciously left quite a bit undone due to sheer lack of time, but each piece left undone has its place planned and the future work required to complete it is queued. We have ranked future contributions to each layer by priority and we provide some explanation of the ones with the most impact in the future work section (7).

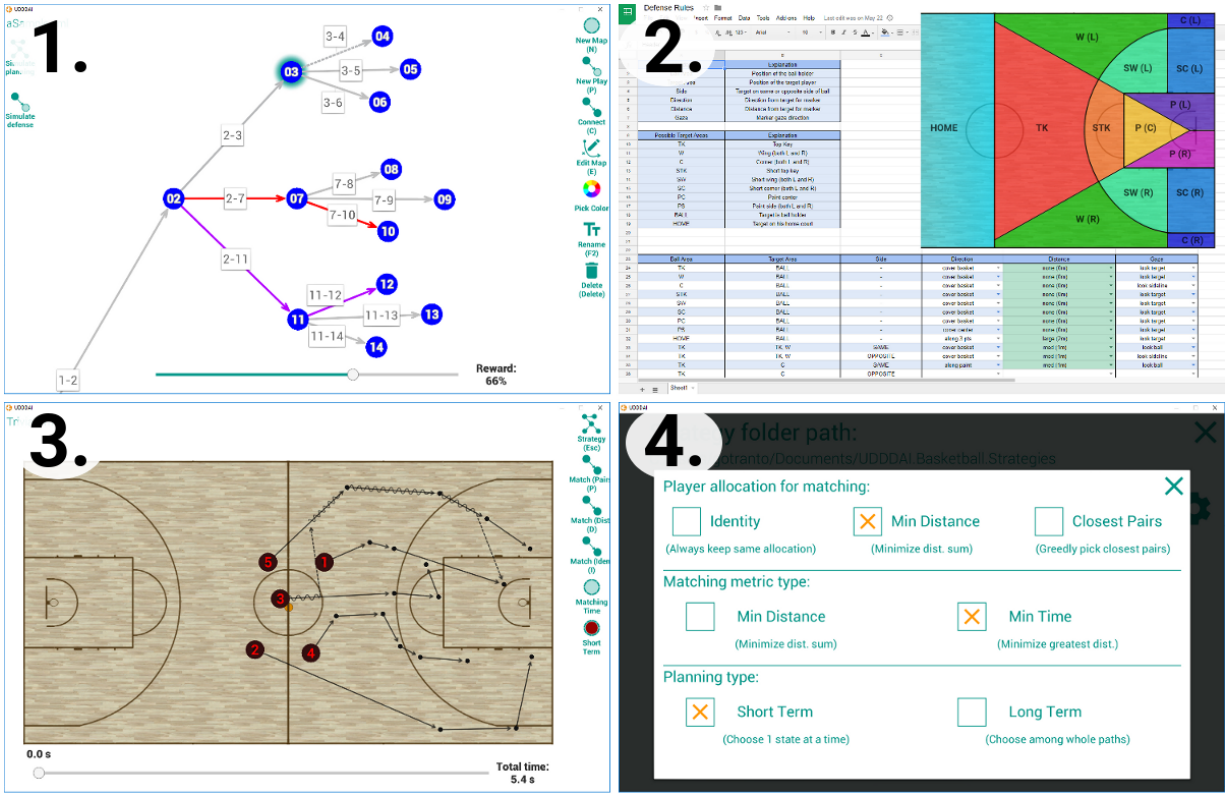


**Figure 6.17:** *Some of the necessary steps to prepare the simulator input. Frame 1: the red team's offensive TSM specification. Frame 2: the blue team's defensive TSM specification. Frame 3: testing the play highlighted in red in frame 1. Frame 4: choosing the simulation preferences.*

Figure 6.17 illustrates the preparation of the simulator input:

1. Offensive TSM design: In this step the user needs to specify the offensive TSM for both teams. In the image we can see the specification of the red team's offensive TSM:
   - Strategy Graph creation: The user specifies the structure of the TSM as well as the risk and reward values for each transition and map, respectively (6.1.1).
   - Strategy Maps creation: The user specifies the equivalence class for each SM of the TSM (6.1.2).
   - Strategy Transitions creation: The user specifies the necessary actions to execute each ST in the TSM (6.1.3).
2. Defensive TSM design: The defensive model design tool is a simple table where the user specifies the desired ideal positioning for each possible relevant match situation. This process is described in more detail in 6.2.
3. Behavior testing: The strategy designer provides tools where the user can test if their strategy behaves the way it is intended. In the image they are testing the play highlighted in red in frame 1. A defense behavior test tool allows them to test the defensive model specified in step 2. There is also a matching and planning test tool where they can ensure that their TSM covers the necessary match situations with adequate responses.
4. Simulation parameters: When starting a new simulation the user can choose the desired simulation parameters. The choices given to the user are:
   (a) Player allocation mode: The choice between fixed, greedy or minimal allocation, as described in 4.2.1.

(b) Matching type: The user can choose between temporal or spacial matching, also described in 4.2.1.

(c) Planning type: The user can choose between short-term planning and long-term planning, as described in 4.1.2.

Figure 6.18 shows a small portion of a simulation in order to illustrate the simulation process. The left side of the image shows a play being executed in the simulation and the right side shows the corresponding point in the ST that the red players are pursuing and the defensive strategy model for blue player 2 (chosen as a representative for the blue team in the figure). There are a few things to note in each frame:

1. Match beginning: Frame 1-A shows the initial position of both teams. The clock is stopped because the user has not yet pushed the play button. Once he does, the tip off occurs after a second. Frame 1-B shows the planning that the red team will try to accomplish. In order to do this the red player will need to win the tip off, which he does, in this case. The blue team switches to defense mode and their players will each be assigned to a red player for marking. The red team will start to execute the play displayed in 1-B. Frame 1-C shows the defensive TSM of the blue player marking red player 2.

2. Red advances: Frame 2-A shows red player 5 with the ball after the tip off. He begins running toward the basket from the left side of the court, his teammates accompany him through their own paths, as shown in 2-B. The blue team reacts to the advance by moving and marking their respective players. The defensive strategy indicating the ideal position each blue player needs to pursue. Note that the user rotated the camera to better see the advancement of the red team. The blue player 2 marking red player 2 still tries to stay between the red player and the basket, as per his defensive model in 2-C.

3. Reaching the 3-pts line: Frame 3-A shows the red player 5 running through the 3-pts line and nearly reaching the paint. As shown in their progress through the play in frame 3-B, red player 1 is nearly at the point where he should receive a pass to end the play with a score attempt. The other players advanced as planned and are in their correct positions. The blue team continued their reactive positioning and are still accompanying the displacement of the red team. The blue player 2 continues to mark red player 2, now looking to the ball and trying to stay between the ball and his marked adversary, as indicated in 3-C.

4. Decision point: In frame 4-A red player 5 attempts a pass to red player 1, who positioned himself under the basket. The blue players marking red players 1 and 5 are both in a position to interfere with the pass, as their TSM dictates. Note from 4-B that the play is nearly at its end and if the pass is successful red player 1 will immediately attempt to score. At this moment in the simulation two outcomes could happen:

   (a) Pass Interception: If any of the two blue players in position to intercept the pass manages to do so, the play of the red team will reach its end. If the blue player keeps a hold of the ball, then the blue team will execute a planning round and the simulation will continue with the blue team attacking. In that case the red team would switch to defensive mode and act according to their defensive TSM.

   (b) Pass successful: If the pass is successful then red player 1 tries to score. If he manages to do so, the play will end and ball possession will be given automatically to the blue team, as per basketball regulation. A blue player will be selected to take the ball outside the court to then serve the ball and start a blue team play. The red team will switch to defensive mode and start protecting their own basket.

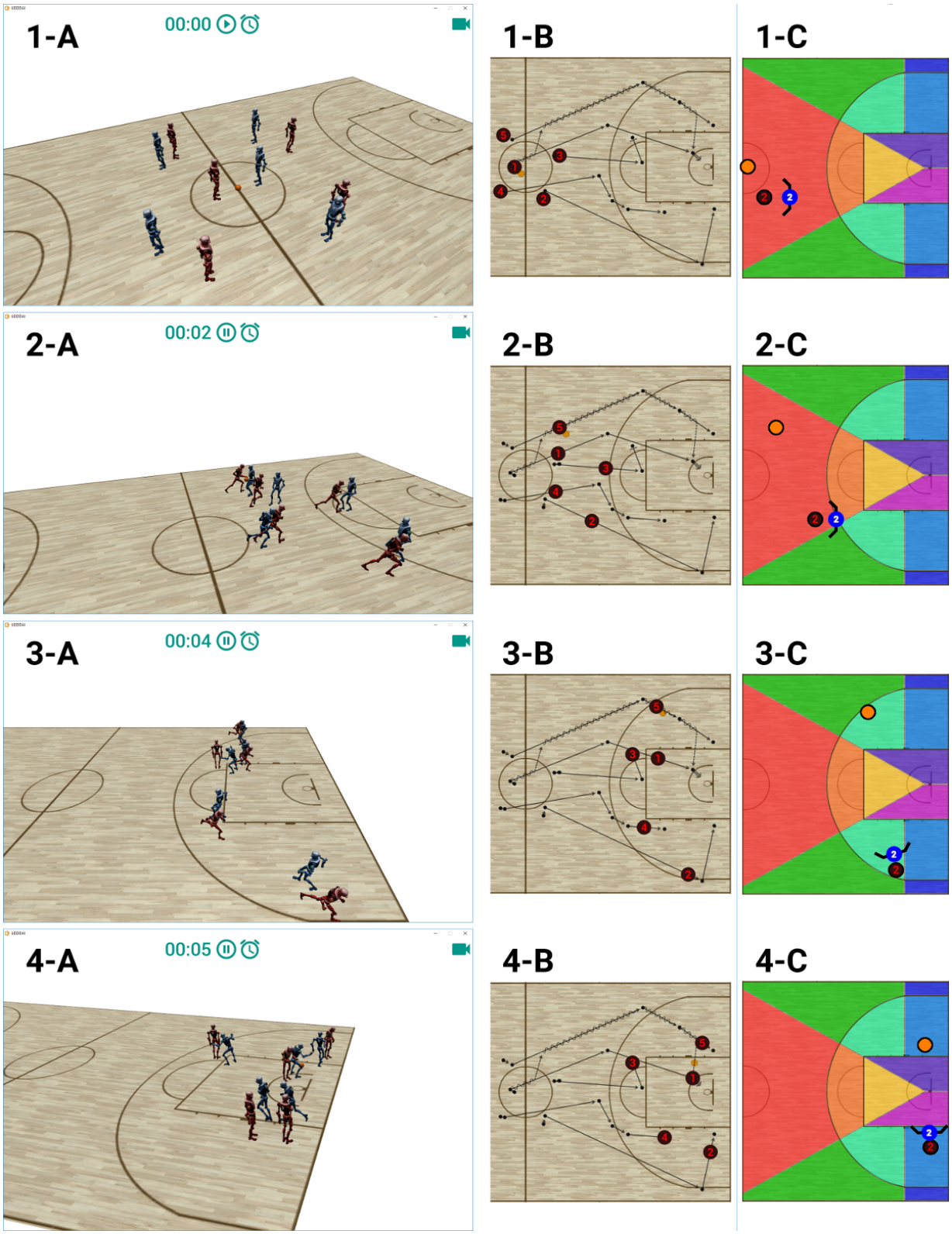


**Figure 6.18:** *The execution of the basketball simulator with the input from 6.17. Column A shows 4 steps of the simulation of a play. Column B shows the corresponding point in the offensive TSM of the red team. Column C shows the corresponding point in the defensive TSM of player 2 in the blue team, marking red player 2.*

# Chapter 7

# Discussion

Our initial ambition for this work was to create new concepts and methods capable of representing and simulating invasion team sports with user defined strategy models. In some ways we believe this original ambition was achieved and surpassed: we defined a formal strategy model and the methods to use it on a general class of domains greater than just invasion team sports. In chapter 3 we defined a formal model to describe strategic behavior that can be used to describe the behavior of a team of agents or just that of a single agent. In chapter 4 we showed how to use our formal model to create high level planners. Finally, we presented in chapter 5 a system architecture that is compatible with our proposed methods and can be used to implement them robustly. We also proved the usability of everything proposed by creating a basketball simulator employing the exact concepts and methods created for this thesis.

Our formal model allows us to calculate plans of actions and control a team of agents in a previously unexplored domain type, while gathering user input to guide the control. This opens the possibility of gathering specialist knowledge in new fields without requiring an immense effort from the specialists to translate their knowledge into an unnatural syntax for their domains. For instance, the design tool we created to gather basketball knowledge from the area's specialists does so using a language already standard in basketball. We believe this insight is important if we aim to create widely available AI tools that follow user input.

We also like the idea of simplifying a domain's representation to the simplest possible format that still represents strategic knowledge. We do so by using equivalence classes, but any format that enables the user to provide structured and applicable knowledge would reap the same benefits. We believe this is a key factor in constructing a bridge between humans and AI systems. A proper application of our methods does require understanding of the domains, but this enables a better connection to the domains' specialists and ultimately a better system.

Our approach to planning allows the controller to think in a simpler domain when calculating plans due to our matching and realization functions. This reduces much of the performance issues that would arise from working in the original domain. Our use of the exploration and exploitation aspects of MCTS allowed us to create a planner that uses the TSM in a faithful manner: exploring the whole model while still prioritizing the best plays. We also managed to create two distinct forms of planning (short-term and long-term) and provided a method to smoothly transition between the two. This insight allows us to effortlessly simulate experience gain for any domain compatible with our system.

Our proposed architecture, described in 5, allowed a single programmer to create a robust and quite sophisticated system from start to finish in a very limited amount of time. The system can control every aspect of the simulation and each level of control is neatly encompassed in their own module. This modularity also facilitates the incorporation of further sophistication in the future, some suggested by us in the next section of this document. On top of the architecture description, we believe the detailed account of our own implementation, provided in chapter 6, can guide the reader through the creation of their own system using of our proposed architecture. We decided to include the mechanical control layer and to ensure that it was compatible with current industry

standards hoping to further increase the adoption of our methodology.

It is not always trivial to see the whole picture, from theory to product implementation, in academic works such as this. We believe that this document is capable of providing such a picture. Not only for our specific product implementation (of a basketball simulator), but many other domains that could be implemented using our work. While we believe this document provides a solid base for the development of simulators, we intend to continue our work and to publish whenever a breakthrough is encountered.

## Future Work

Our work covered a wide range of topics, from sports theory, through AI and reaching high fidelity graphical simulations. It is natural that we could not explore every improvement idea that emerged during the development of this thesis. However, we did register plans for the development of the most promising ideas. What follows in this section is a list of these with a short explanation of the rationale that lead us to believe working on them would yield good results. We present them in no particular order.

### Evolving from a collective brain to autonomous agents

As we explained in 3.1.4, there is an equivalent model representation that transitions the view from a collective brain to individual autonomous agents. The system sophistications we have mapped out for the basketball simulator using this representation are:

1. The collective planner should use the TSM representation of the player with the ball. This representation could be different for each player, yielding different results depending on which player is the planner.
2. Once the planner decided on a play, he needs to communicate it to the rest of the team. This is interesting because:
   (a) There could be a lag in the communication, which would cause some players to start the play execution late.
   (b) There could be errors in the message that causes some players to execute different plays until the message can be corrected.
   (c) The play in each player's TSM could be different, causing each to execute their own version of the play.
3. If each agent perceives the world differently, based on different view angles or even sensor discrepancies, their execution of the play could change to match the new perceptions.

Partially observable domains have been thoroughly covered in planning literature and many of the necessary concepts of sensor use and partial environment representation have already been studied ([Cas98], [CCO+12]).

### Strategy graph structure study

As mentioned in the definition of the strategy graph (3.1.3), the structure of the strategy graph can carry interesting information for each domain. Graph structure has been studied quite a bit in literature, but our domain specific instances of the strategy graph could bring new insight into how different characteristics impact the underlining behavior. The obvious observation that a greater number of states and edges will lead to more complex behavior could be improved until it actually brings new insights. We believe Game Theory ([FT91]) can provide an adequate framework to support the analysis of the competition between two opposing strategies with different graphs.

We also mentioned the possibility of learning the strategy graph's risk and reward values through simulation in 3.2.3. Of course simply learning the risk and reward values would be only one of the possible steps in the strategy graph improvements. A sophisticated simulator could give coaches insights on which plays would work better against specific defense strategies, for instance.

### Real time agent control by the user

Our system can be adapted to allow the user to directly control an agent in the simulation. This is constantly done in video games where the user controls one of the agents of his team while the others move automatically. The controlled agent can be the same through the simulation or it could change (e.g., the user could always take control of the player with the ball in a basketball match).

There are solutions available in the market that enable us to acquire real time body poses of a person ([OKA11]). This poses can be transported into the simulator to enable the user to directly control an agent's body. In order to truly immerse the user in the domain, we could easily integrate our camera system to work with a virtual reality headset that renders exactly what the agent is seeing to the display of the user. A possible application of this system would be to train a virtual goalkeeper to position himself correctly to react do different situation. His unique field of view through the virtual reality headset would reflect a real situation more adequately than a simple screen rendering ever could.

### Other domains

There is an obvious opportunity to work with the proposed system in different domains. We have already accomplished some results working with the soccer domain in some capacity, usually match analysis. The proposed system is generic enough to accommodate a large number of different domains and we are curious to see the kind of results we can accomplish with it in the future.

### More input control

We simplified our work up to this point by only creating user-facing design tool for the offensive TSM. Our system allows for the input of a defensive TSM as well as player models for each player of both teams. In order to be able to accept user generated input for these, we need to create a design tool that enables the user to generate the necessary data.

## Our publications

We have published three papers so far: [LSOB14], [LBOU14] and [LOB16]. We have also submitted a fourth for publication: [LDOBed].

# Bibliography


[AK01] Amin Atrash and Sven Koenig. Probabilistic planning for behavior-based robots. In *FLAIRS Conference*, pages 531–535, 2001. 11

[Alt99] Eitan Altman. *Constrained Markov decision processes*, volume 7. CRC Press, 1999. 11

[BA98] Tucker Balch and Ronald C Arkin. Behavior-based formation control for multirobot teams. *IEEE transactions on robotics and automation*, 14(6):926–939, 1998. 30

[BBS95] Andrew G Barto, Steven J Bradtke and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial intelligence*, 72(1-2):81–138, 1995. 13

[BDG00] Craig Boutilier, Richard Dearden and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial intelligence*, 121(1):49–107, 2000. 12

[Bel13] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013. 13

[BF97] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997. 1, 10

[BG03] Blai Bonet and Hector Geffner. Labeled rtdp: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003. 13

[BJM02] Andrew Borrie, Gudberg K Jonsson and Magnus S Magnusson. Temporal pattern analysis and its applicability in sport: an explanation and exemplar data. *Journal of sports sciences*, 20(10):845–852, 2002. 24, 25

[BL97] Avrim L Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1):245–271, 1997. 34

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012. 35

[Cas98] Anthony Rocco Cassandra. Exact and approximate algorithms for partially observable markov decision processes. 1998. 5, 80

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008. 35

[CCO+12] A. Coles, A. Coles, A. García Olaya, S. Jiménez, C. Linares López, S. Sanner and S. Yoon. A survey of the seventh international planning competition. *Artificial Intelligence Magazine (AI Magazine)*, 33(1):83–88, 2012. 5, 80

[CF98] Cristiano Castelfranchi and Rino Falcone. Towards a theory of delegation for agent-based systems. *Robotics and Autonomous Systems*, 24(3-4):141–157, 1998. 24

[CMDO16] Michele Colledanchise, Alejandro Marzinotto, Dimos V Dimarogonas and Petter Oegren. The advantages of using behavior trees in mult-robot systems. In *ISR 2016: 47st International Symposium on Robotics; Proceedings of*, pages 1–8. VDE, 2016. 15

[CO14] Michele Colledanchise and Petter Ögren. How behavior trees modularize robustness and safety in hybrid systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1482–1488. IEEE, 2014. 16

[Cou07] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *Lecture Notes in Computer Science*, 4630:72–83, 2007. 35

[CYS94] Randall J Calistri-Yeh and Alberto Maria Segre. The design of alps: An adaptive learning and planning system. In *AIPS*, pages 207–212, 1994. 41

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. 8

[FC01] Rino Falcone and Cristiano Castelfranchi. The human in the loop of a delegated agent: The theory of adjustable social autonomy. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 31(5):406–418, 2001. 24

[FL03] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124, 2003. 6, 61

[FLB+13] David Ferrucci, Anthony Levas, Sugato Bagchi, David Gondek and Erik T Mueller. Watson: beyond jeopardy! *Artificial Intelligence*, 199:93–105, 2013. 38

[FN71] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971. 6

[FN93] Richard E Fikes and Nils J Nilsson. Strips, a retrospective. *Artificial Intelligence*, 59(1):227–232, 1993. 6

[FT91] Drew Fudenberg and Jean Tirole. Game theory, 1991. *Cambridge, Massachusetts*, 393:12, 1991. 80

[GBD97] Jean-Francis Grehaigne, Daniel Bouthier and Bernard David. Dynamic-system analysis of opponent relationships in collective actions in soccer. *Journal of Sports Sciences*, 15(2):137–149, 1997. 24, 25

[GCG10] Jean-Francis Gréhaigne, Didier Caty and Paul Godbout. Modelling ball circulation in invasion team sports: a way to promote learning games through understanding. *Physical Education and Sport Pedagogy*, 15(3):257–270, 2010. 24, 25

[GKPV03] Carlos Guestrin, Daphne Koller, Ronald Parr and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research*, 19:399–468, 2003. 12

[GL05] Alfonso Gerevini and Derek Long. Plan constraints and preferences in pddl3. *The Language of the Fifth International Planning Competition. Tech. Rep. Technical Report, Department of Electronics for Automation, University of Brescia, Italy*, 75, 2005. 6

[GLPH15] Kelleher R Guerin, Colin Lea, Chris Paxton and Gregory D Hager. A framework for end-user instruction of a robot assistant for manufacturing. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 6167–6174. IEEE, 2015. 15, 16

[GM85] Michael Girard and Anthony A Maciejewski. Computational modeling for the computer animation of legged figures. In *ACM SIGGRAPH Computer Graphics*, volume 19, pages 263–270. ACM, 1985. 18

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos and Olivier Teytaud. *Modification of UCT with patterns in Monte-Carlo Go*. PhD Thesis, INRIA, 2006. 35

[HNR68] Peter E Hart, Nils J Nilsson and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 8

[HP68] Fredrick J Hill and Gerald R Peterson. *Introduction to switching theory and logic design*. Wiley, 1968. 20

[HTD90] James A Hendler, Austin Tate and Mark Drummond. Ai planning: Systems and techniques. *AI magazine*, 11(2):61, 1990. 5

[Isl05] Damian Isla. Managing complexity in the halo2 ai. In *Game Developer's Conference Proceedings, 2005*, 2005. 15

[JGT14] Emil Juul Jacobsen, Rasmus Greve and Julian Togelius. Monte mario: platforming with mcts. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 293–300. ACM, 2014. 35

[KG03] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 214–224. Eurographics Association, 2003. 17

[KP00] Daphne Koller and Ronald Parr. Policy iteration for factored mdps. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 326–334. Morgan Kaufmann Publishers Inc., 2000. 12, 13

[KS+92] Henry A Kautz, Bart Selman et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363, 1992. 10

[KT12] Sergey Karakovskiy and Julian Togelius. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012. 15

[KTS+98] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda and Minoru Asada. The robocup synthetic agent challenge 97. In *RoboCup-97: Robot Soccer World Cup I*, pages 62–73. Springer, 1998. 13

[LaV06] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006. 6, 9

[LBC10] Chong-U Lim, Robin Baumgarten and Simon Colton. Evolving behaviour trees for the commercial game defcon. In *European Conference on the Applications of Evolutionary Computation*, pages 100–110. Springer, 2010. 15

[LBC+12] Patrick Lucey, Alina Bialkowski, Peter Carr, Eric Foote and Iain A Matthews. Characterizing multi-agent team behavior from partial team tracings: Evidence from the english premier league. In *AAAI*, 2012. 24

[LBOU14] Leonardo Lamas, Junior Barrera, Guilherme Otranto and Carlos Ugrinowitsch. Invasion team sports: strategy and match modeling. *International Journal of Performance Analysis in Sport*, 14(1):307–329, 2014. 1, 2, 81

[LDOBed] Leonardo Lamas, Rene Drezner, Guilherme Otranto and Junior Barrera. Analytic method for evaluating players' decisions in team sports, 2017. submitted. 81

[LGM98] Michael L Littman, Judy Goldsmith and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9(1):1–36, 1998. 12

[LJS+11] L Lamas, D De Rose Junior, F Santana, E Rostaiser, L Negretti and C Ugrinowitsch. Space creation dynamics in basketball offence: validation and evaluation of elite teams. *International Journal of Performance Analysis in Sport*, 11(1):71–84, 2011. 25

[LOB16] Leonardo Lamas, Guilherme Otranto and Junior Barrera. Computational system for strategy design and match simulation in team sports. In *Proceedings of the 10th International Symposium on Computer Science in Sports (ISCSS)*, pages 69–76. Springer, 2016. 2, 81

[LSOB14] Leonardo Lamas, Felipe Santana, Guilherme Otranto and Junior Barrera. Inference of team sports strategies based on a library of states: application to basketball. In *Proceedings of the 2014 KDD Workshop on Large-Scale Sports Analytics*, 2014. 2, 24, 81

[LSR05] Tim Laue, Kai Spiess and Thomas Röfer. Simrobot-a general physical robot simulator and its application in robocup. In *RoboCup*, pages 173–183. Springer, 2005. 14

[McD96] Drew V McDermott. A heuristic estimator for means-ends analysis in planning. In *AIPS*, volume 96, pages 142–149, 1996. 10

[MF16] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2016. 14

[MGH+98] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld and David Wilkins. Pddl-the planning domain definition language. 1998. 6

[NPLOB16] Miguel Nicolau, Diego Perez-Liebana, Michael O'Neill and Anthony Brabazon. Evolutionary behavior tree approaches for navigating platform games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2016. 24

[OKA11] Iason Oikonomidis, Nikolaos Kyriazis and Antonis A Argyros. Efficient model-based 3d tracking of hand articulations using kinect. In *BmVC*, volume 1, page 3, 2011. 17, 81

[PNOB11] Diego Perez, Miguel Nicolau, Michael O'Neill and Anthony Brabazon. Reactiveness and navigation in computer games: Different needs, different approaches. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 273–280. IEEE, 2011. 15

[Poh69] Ira Pohl. *Bi-directional and heuristic search in path problems*. PhD Thesis, Department of Computer Science, Stanford University, 1969. 8

[PTCd05] Sébastien Paquet, Ludovic Tobin and Brahim Chaib-draa. An online pomdp algorithm used by the policeforce agents in the robocuprescue simulation. In *Robot Soccer World Cup*, pages 196–207. Springer, 2005. 14

[Put14] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014. 11

[QMSW16] Jiahu Qin, Qichao Ma, Yang Shi and Long Wang. Recent advances in consensus of multi-agent systems: A brief survey. *IEEE Transactions on Industrial Electronics*, 2016. 24

[Rin12] Jussi Rintanen. Planning as satisfiability: Heuristics. *Artificial Intelligence*, 193:45–86, 2012. 10

[RN09] Stuart Jonathan Russell and Peter Norvig. Artificial intelligence: a modern approach (3rd edition), 2009. 5, 6, 8, 13

[Ros97] Julio K Rosenblatt. Damn: A distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):339–360, 1997. 48

[RRP12] Bobby Bodenheimer Chuck Rose, Seth Rosenthal and John Pella. The process of motion capture: Dealing with the data. In *Computer Animation and Simulation'97: Proceedings of the Eurographics Workshop in Budapest, Hungary, September 2–3, 1997*, page 3. Springer, 2012. 17

[SA88] David Slate and Larry Atkin. Chess 4.5: The northwestern university chess program. In *Computer chess compendium*, pages 80–103. Springer, 1988. 8

[SD06] Fernando Seabra and Luis EPBT Dantas. Space definition for match analysis in soccer. *International Journal of Performance Analysis in Sport*, 6(2):97–113, 2006. 24, 25

[SGJ+11] Alexander Shoulson, Francisco Garcia, Matthew Jones, Robert Mead and Norman Badler. Parameterizing behavior trees. *Motion in Games*, pages 144–155, 2011. 15

[SPF03] Ari Shapiro, Fred Pighin and Petros Faloutsos. Hybrid control for interactive character animation. In *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, pages 455–461. IEEE, 2003. 18

[THRB10] Albin Tenga, Ingar Holme, Lars Tore Ronglan and Roald Bahr. Effect of playing tactics on achieving score-box possessions in a random series of team possessions from norwegian professional soccer matches. *Journal of sports sciences*, 28(3):245–255, 2010. 25

[Wel93] Chris Welman. *Inverse kinematics and geometric constraints for articulated figure manipulation*. PhD Thesis, Theses (School of Computing Science)/Simon Fraser University, 1993. 18

[WSN+17] Shir Li Wang, Kamran Shafi, Theam Foo Ng, Chris Lokan and Hussein A Abbass. Contrasting human and computational intelligence based autonomous behaviors in a blue–red simulation environment. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(1):27–40, 2017. 1